

# Probabilistic Recursion Theory and Implicit Complexity\*

Ugo Dal Lago<sup>†</sup>

Sara Zuppiroli<sup>‡</sup>

June 26, 2014

## Abstract

We show that probabilistic computable functions, i.e., those functions outputting distributions and computed by probabilistic Turing machines, can be characterized by a natural generalization of Church and Kleene's partial recursive functions. The obtained algebra, following Leivant, can be restricted so as to capture the notion of polytime sampleable distributions, a key concept in average-case complexity and cryptography.

## 1 Introduction

Models of computation as introduced one after the other in the first half of the last century were all designed around the assumption that *determinacy* is one of the key properties to be modeled: given an algorithm and an input to it, the sequence of computation steps leading to the final result is *uniquely* determined by the way an *algorithm* describes the state evolution. The great majority of the introduced models are *equivalent*, in that the classes of functions (on, say, natural numbers) they are able to compute are the same.

The second half of the 20th century has seen the assumption above relaxed in many different ways. Nondeterminism, as an example, has been investigated as a way to abstract the behavior of certain classes of algorithms, this way facilitating their study without necessarily changing their expressive power: think about how NFAs make the task of proving closure properties of regular languages easier [15].

A relatively recent step in this direction consists in allowing algorithms' internal state to evolve probabilistically: the next state is not *functionally* determined by the current one, but is obtained from it by performing a process having possibly many outcomes, each with a certain probability. Again, probabilistically evolving computation can be a way to abstract over determinism, but also a way to model situations in which algorithms have access to a source of true randomness.

Probabilistic models are nowadays more and more pervasive. Not only are they a formidable tool when dealing with uncertainty and incomplete information, but they sometimes are a *necessity* rather than an option, like in computational cryptography (where, e.g., secure public key encryption schemes need to be probabilistic [9]). A nice way to deal computationally with probabilistic models is to allow probabilistic choice as a primitive when designing algorithms, this way switching from usual, deterministic computation to a new paradigm, called probabilistic computation. Examples of application areas in which probabilistic computation has proved to be useful include natural language processing [?], robotics [?], computer vision [?], and machine learning [?].

But what does the presence of probabilistic choice give us in terms of expressivity? Are we strictly more expressive than usual, deterministic, computation? And how about efficiency: is it that probabilistic choice permits to solve computational problems more efficiently? These questions have been among the most central in the theory of computation, and in particular in

---

\*This work is partially supported by the ANR project 12IS02001 PACE

<sup>†</sup>Università di Bologna & INRIA, [dallago@cs.unibo.it](mailto:dallago@cs.unibo.it)

<sup>‡</sup>Università di Bologna, [zuppiroli@cs.unibo.it](mailto:zuppiroli@cs.unibo.it)

computational complexity, in the last forty years (see below for more details about related work). Roughly, while probability has been proved not to offer any advantage in the absence of resource constraints, it is not known whether probabilistic classes such as **BPP** or **ZPP** are different from **P**.

This work goes in a somehow different direction: we want to study probabilistic computation without necessarily *reducing* or *comparing* it to deterministic computation. The central assumption here is the following: a probabilistic algorithm computes what we call a *probabilistic function*, i.e. a function from a discrete set (e.g. natural numbers or binary strings) to *distributions* over the same set. What we want to do is to study the set of those probabilistic functions which can be computed by algorithms, possibly with resource constraints.

We give some initial results here. First of all, we provide a characterization of computable probabilistic functions by the natural generalization of Kleene’s partial recursive functions, where among the initial functions there is now a function corresponding to tossing a fair coin. In the non-trivial proof of completeness for the obtained algebra, Kleene’s minimization operator is used in an unusual way, making the usual proof strategy for Kleene’s Normal Form Theorem (see, e.g., [18]) useless. We later hint at how to recover the latter by replacing minimization with a more powerful operator. We also mention how probabilistic recursion theory offers characterizations of concepts like the one of a computable distribution and of a computable real number.

The second part of this paper is devoted to applying the aforementioned recursion-theoretical framework to polynomial-time computation. We do that by following Bellantoni and Cook’s and Leivant’s works [1, 12], in which polynomial-time deterministic computation is characterized by a restricted form of recursion, called *predicative* or *ramified* recursion. Endowing Leivant’s ramified recurrence with a random base function, in particular, is shown to provide a characterization of polynomial-time computable distributions, a key notion in average-case complexity [2].

**Related Work.** This work is rooted in classic theory of computation, and in particular in the definition of partial computable functions as introduced by Church and later studied by Kleene [11]. Starting from the early fifties, various forms of automata in which probabilistic choice is available have been considered (e.g. [14]). The inception of probabilistic choice into an universal model of computation, namely Turing machines, is due to Santos [16, 17], but is (essentially) already there in an earlier work by De Leeuw and others [5]. Some years later, Gill [6] considered probabilistic Turing machines with bounded complexity: his work has been the starting point of a florid research about the interplay between computational complexity and randomness. Among the many side effects of this research one can of course mention modern cryptography [10], in which algorithms (e.g. encryption schemes, authentication schemes, and adversaries for them) are very often assumed to work in probabilistic polynomial time.

Implicit computational complexity (ICC), which studies machine-free characterizations of complexity classed based on mathematical logic and programming language theory, is a much younger research area. Its birth is traditionally made to correspond with the beginning of the nineties, when Bellantoni and Cook [1] and Leivant [12] independently proposed function algebras precisely characterizing (deterministic) polynomial time computable functions. In the last twenty years, the area has produced many interesting results, and complexity classes spanning from the logarithmic space computable functions to the elementary functions have been characterized by, e.g., function algebras, type systems [13], or fragments of linear logic [7]. Recently, some investigations on the interplay between implicit complexity and probabilistic computation have started to appear [3]. There is however an intrinsic difficulty in giving *implicit* characterizations of probabilistic classes like **BPP** or **ZPP**: the latter are semantic classes defined by imposing a polynomial bound on time, but also appropriate bounds on the probability of error. This makes the task of enumerating machines computing problems in the classes much harder and, ultimately, prevents from deriving implicit characterization of the classes above. Again, our emphasis is different: we do not see probabilistic algorithms as artifacts computing functions of the same kind as the one deterministic algorithms compute, but we see probabilistic algorithms as devices outputting distributions.

## 2 Probabilistic Recursion Theory

In this section we provide a characterization of the functions computed by a Probabilistic Turing Machine (PTM) in terms of a function algebra *à la* Kleene. We first define *probabilistic recursive functions*, which are the elements of our algebra. Next we define formally the class of probabilistic functions computed by a PTM. Finally, we show the equivalence of the two introduced classes. In the following,  $\mathbb{R}_{[0,1]}$  is the unit interval.

Since PTMs compute probability (pseudo-)distributions, the functions that we consider in our algebra have domain  $\mathbb{N}^k$  and codomain  $\mathbb{N} \rightarrow \mathbb{R}_{[0,1]}$  (rather than  $\mathbb{N}$  as in the classic case). The idea is that if  $f(x)$  is a function which returns  $r \in \mathbb{R}_{[0,1]}$  on input  $y \in \mathbb{N}$ , then  $r$  is the probability of getting  $y$  as the output when feeding  $f$  with the input  $x$ . We note that we could extend our codomain from  $\mathbb{N} \rightarrow \mathbb{R}_{[0,1]}$  to  $\mathbb{N}^m \rightarrow \mathbb{R}_{[0,1]}$ , however we use  $\mathbb{N} \rightarrow \mathbb{R}_{[0,1]}$  in order to simplify the presentation.

**Definition 1 (Pseudodistributions and Probabilistic Functions)** A pseudodistribution on  $\mathbb{N}$  is a function  $\mathcal{D} : \mathbb{N} \rightarrow \mathbb{R}_{[0,1]}$  such that  $\sum_{n \in \mathbb{N}} \mathcal{D}(n) \leq 1$ .  $\sum_{n \in \mathbb{N}} \mathcal{D}(n)$  is often denoted as  $\sum \mathcal{D}$ . Let  $\mathbb{P}_{\mathbb{N}}$  be the set of all pseudodistributions on  $\mathbb{N}$ . A probabilistic function (PF) is a function from  $\mathbb{N}^k$  to  $\mathbb{P}_{\mathbb{N}}$ , where  $\mathbb{N}^k$  stands for the set of  $k$ -tuples in  $\mathbb{N}$ . We use the expression  $\{n_1^{p_1}, \dots, n_k^{p_k}\}$  to denote the pseudodistribution  $\mathcal{D}$  defined as  $\mathcal{D}(n) = \sum_{n_i = n} p_i$ . Observe that  $\sum \mathcal{D} = \sum_{k=1}^k p_i$ .

Please notice that probabilistic functions are always *total* functions, but their codomain is a set of distributions which do not necessarily sum to 1, but rather to a real number *smaller* or equal to 1, this way modeling the probability of divergence. For example, the nowhere-defined partial function  $\Omega : \mathbb{N} \rightarrow \mathbb{N}$  of classic recursion theory becomes a probabilistic function which returns the empty distributions  $\emptyset$  on any input. The first step towards defining our function algebra consists in giving a set of functions to start from:

**Definition 2 (Basic Probabilistic Functions)** The basic probabilistic functions (BPFs) are as follows:

- The zero function  $z : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  defined as:  $z(n)(0) = 1$  for every  $n \in \mathbb{N}$ ;
- The successor function  $s : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  defined as:  $s(n)(n+1) = 1$  for every  $n \in \mathbb{N}$ ;
- The projection function  $\Pi_m^n : \mathbb{N}^n \rightarrow \mathbb{P}_{\mathbb{N}}$  defined as:  $\Pi_m^n(k_1, \dots, k_n)(k_m) = 1$  for every positive  $n, m \in \mathbb{N}$  such that  $1 \leq m \leq n$ ;
- The fair coin function  $r : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  that is defined as:

$$r(x)(y) = \begin{cases} 1/2 & \text{if } y = x \\ 1/2 & \text{if } y = x + 1 \end{cases}$$

The first three BPFs are the same as the basic functions from classic recursion theory, while  $r$  is the only truly probabilistic BPF.

The next step consists in defining how PFs *compose*. Function composition of course cannot be used here, because when composing two PFs  $g$  and  $f$  the codomain of  $g$  does not match with the domain of  $f$ . Indeed  $g$  returns a distribution  $\mathbb{N} \rightarrow \mathbb{R}_{[0,1]}$  while  $f$  expects a natural number as input. What we have to do here is the following. Given an input  $x \in \mathbb{N}$  and an output  $y \in \mathbb{N}$  for the composition  $f \bullet g$ , we apply the distribution  $g(x)$  to any value  $z \in \mathbb{N}$ . This gives a probability  $g(x)(z)$  which is then multiplied by the probability that the distribution  $f(z)$  associates to the value  $y \in \mathbb{N}$ . If we then consider the sum of the obtained product  $g(x)(z) \cdot f(z)(y)$  on all possible  $z \in \mathbb{N}$  we obtain the probability of  $f \bullet g$  returning  $y$  when fed with  $x$ . The sum is due to the fact that two different values, say  $z_1, z_2 \in \mathbb{N}$ , which provide two different distributions  $f(z_1)$  and  $f(z_2)$  must both contribute to the same probability value  $f(z_1)(y) + f(z_2)(y)$  for a specific  $y$ . In other words, we are doing nothing more than lifting  $f$  to a function from distributions to distributions, then composing it with  $g$ . Formally:

**Definition 3 (Composition)** We define the composition  $f \bullet g : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  of two functions  $f : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  and  $g : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  as:

$$((f \bullet g)(x))(y) = \sum_{z \in \mathbb{N}} g(x)(z) \cdot f(z)(y).$$

The previous definition can be generalized to functions taking more than one parameter in the expected way:

**Definition 4 (Generalized Composition)** *We define the generalized composition of functions  $f : \mathbb{N}^n \rightarrow \mathbb{P}_{\mathbb{N}}$ ,  $g_1 : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$ ,  $\dots$ ,  $g_n : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$  as the function  $f \odot (g_1, \dots, g_n) : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$  defined as follows:*

$$((f \odot (g_1, \dots, g_n))(\mathbf{x}))(y) = \sum_{z_1, \dots, z_n \in \mathbb{N}} \left( f(z_1, \dots, z_n)(y) \cdot \prod_{1 \leq i \leq n} g_i(\mathbf{x})(z_i) \right).$$

With a slight abuse of notation, we can treat probabilistic functions as ordinary functions when forming expressions. Suppose, as an example, that  $x \in \mathbb{N}$  and that  $f : \mathbb{N}^3 \rightarrow \mathbb{P}_{\mathbb{N}}$ ,  $g : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$ ,  $h : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$ . Then the expression  $f(g(x), x, h(x))$  stands for the distribution in  $\mathbb{P}_{\mathbb{N}}$  defined as follows:  $(f \odot (g, id, h))(x)$ , where  $id = \Pi_1^1$  is the identity PF.

The way we have defined probabilistic functions and their composition is reminiscent of, and indeed inspired by, the way one defines the Kleisli category for the Girmonad, starting from the category of partial functions on sets. This categorical way of seeing the problem can help a lot in finding the right definition, but by itself is not adequate to proving the existence of a correspondence with machines like the one we want to give here.

Primitive recursion is defined as in Kleene's algebra, provided that one uses composition as previously defined:

**Definition 5 (Primitive Recursion)** *Given functions  $g : \mathbb{N}^{k+2} \rightarrow \mathbb{P}_{\mathbb{N}}$ , and  $f : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$ , the function  $h : \mathbb{N}^{k+1} \rightarrow \mathbb{P}_{\mathbb{N}}$  defined as*

$$h(\mathbf{x}, 0) = f(\mathbf{x}); \quad h(\mathbf{x}, y + 1) = g(\mathbf{x}, y, h(\mathbf{x}, y));$$

*is said to be defined by primitive recursion from  $f$  and  $g$ , and is denoted as  $rec(f, g)$ .*

We now turn our attention to the minimization operator which, as in the deterministic case, is needed in order to obtain the full expressive power of (P)TMs. The definition of this operator is in our case delicate and requires some explanation. Recall that, in the classic case, the minimization operator allows from a partial function  $f : \mathbb{N}^{k+1} \rightarrow \mathbb{N}$ , to define another partial function, call it  $\mu f$ , which computes from  $\mathbf{x} \in \mathbb{N}^k$  the least value of  $y$  such that  $f(\mathbf{x}, y)$  is equal to 0, if such a value exists (and is undefined otherwise). In our case, again, we are concerned with distributions, hence we cannot simply consider the least value on which  $f$  returns 0, since functions return 0 *with a certain probability*. The idea is then to define the minimization  $\mu f$  as a function which, given an input  $\mathbf{x} \in \mathbb{N}^k$ , returns a distribution associating to each natural  $y$  the probability that the result of  $f(\mathbf{x}, y)$  is 0 and the result of  $f(\mathbf{x}, z)$  is positive for every  $z < y$ . Formally:

**Definition 6 (Minimization)** *Given a PF  $f : \mathbb{N}^{k+1} \rightarrow \mathbb{P}_{\mathbb{N}}$ , we define another PF  $\mu f : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$  as follows:*

$$\mu f(\mathbf{x})(y) = f(\mathbf{x}, y)(0) \cdot \left( \prod_{z < y} \left( \sum_{k > 0} f(\mathbf{x}, z)(k) \right) \right).$$

We are finally able to define the class of functions we are interested in as follows.

**Definition 7 (Probabilistic Recursive Functions)** *The class  $\mathcal{PR}$  of probabilistic recursive functions is the smallest class of probabilistic functions that contains the BPFs (Definition 2) and is closed under the operation of General Composition (Definition 4), Primitive Recursion (Definition 5) and Minimization (Definition 6).*

It is easy to show that  $\mathcal{PR}$  includes all partial recursive functions. This can be done by first defining an extended Recursive Function as follows.

**Definition 8 (Extended Recursive Functions)** For every partial recursive function  $f : \mathbb{N}^k \rightarrow \mathbb{N}$  we define as the extended function  $p_f : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$  as follows:

$$p_f(\mathbf{x})(y) = \begin{cases} 1 & \text{if } y = f(\mathbf{x}) \\ 0 & \text{otherwise} \end{cases}$$

**Proposition 1** If  $f$  is a Partial Recursive function then we define  $p_f$  as defined above is in  $\mathcal{PR}$ .

**Proof.** The proof goes by induction on the structure of  $f$  as a partial recursive function.

- $f$  is the zero function, so  $f : \mathbb{N} \rightarrow \mathbb{N}$  defined as:  $f(x) = 0$  for every  $x \in \mathbb{N}$ . Thus  $p_f$  is in  $\mathcal{PR}$  because  $p_f = z$
- $f$  is the successor function  $s : \mathbb{N} \rightarrow \mathbb{N}$  defined as:  $s(x) = x + 1$  for every  $x \in \mathbb{N}$ . Thus  $p_f$  is in  $\mathcal{PR}$  because  $p_f = s$
- $f$  is the projection function.  $f_m^n : \mathbb{N}^n \rightarrow \mathbb{N}$  defined as:  $f_m^n(x_1, \dots, x_n) = x_m$  for every positive  $n \in \mathbb{N}$  and for all  $m \in \mathbb{N}$ , such that  $1 \leq m \leq n$ . Thus  $p_f$  is in  $\mathcal{PR}$  because  $p_f = \Pi_m^n$
- $f$  is defined by composition from  $h, g_1, \dots, g_n$  as:

$$f(\mathbf{x}) = h(g_1(\mathbf{x}), \dots, g_n(\mathbf{x}))$$

where  $h : \mathbb{N}^n \rightarrow \mathbb{N}$  and  $g_i : \mathbb{N}^k \rightarrow \mathbb{N}$  for every  $1 \leq i \leq n$  are partial recursive functions. by definition of  $f(\mathbf{x})$  we have

$$p_f(\mathbf{x})(y) = \begin{cases} 1 & \text{if } y = h(g_1(\mathbf{x}), \dots, g_n(\mathbf{x})) \\ 0 & \text{otherwise} \end{cases}$$

We see that  $h, g_1, \dots, g_n$  are all partial recursive functions. So we have by definition of  $p_f$  that

$$\begin{aligned} p_{g_1}(\mathbf{x})(y) &= \begin{cases} 1 & \text{if } y = g_1(\mathbf{x}) \\ 0 & \text{otherwise} \end{cases} \\ &\vdots \\ p_{g_n}(\mathbf{x})(y) &= \begin{cases} 1 & \text{if } y = g_n(\mathbf{x}) \\ 0 & \text{otherwise} \end{cases} \\ p_h(\mathbf{z})(y) &= \begin{cases} 1 & \text{if } y = h(\mathbf{z}) \\ 0 & \text{otherwise} \end{cases} \end{aligned}$$

By hypothesis we observe that  $p_{g_1}, \dots, p_{g_n}, p_h \in \mathcal{PR}$  and

$$\begin{aligned} ((p_h \odot (p_{g_1}, \dots, p_{g_n}))(\mathbf{x}))(y) &= \sum_{z_1, \dots, z_n \in \mathbb{N}} p_h(z_1, \dots, z_n)(y) \cdot \left( \prod_{1 \leq i \leq n} p_{g_i}(\mathbf{x})(z_i) \right) \\ &= \sum_{z_1, \dots, z_n \in \mathbb{N}} p_h(z_1, \dots, z_n)(y) \cdot \left( \prod_{z_i = g_i(\mathbf{x})} 1 \right) \\ &= \sum_{y = h(z_1, \dots, z_n)} 1 \cdot \left( \prod_{z_i = g_i(\mathbf{x})} 1 \right) \\ &= \sum_{y = h(g_1(\mathbf{x}), \dots, g_n(\mathbf{x}))} 1 \\ &= p_f(\mathbf{x})(y) \end{aligned}$$

Thus  $p_f$  is in  $\mathcal{PR}$  because  $p_f = ((p_h \odot (p_{g_1}, \dots, p_{g_n}))(\mathbf{x}))(y)$ .

- $f$  is defined by primitive recursion so  $f : \mathbb{N}^k \times \mathbb{N} \rightarrow \mathbb{N}$  defined as:

$$f(\mathbf{x}, 0) = h(\mathbf{x})$$

$$f(\mathbf{x}, n+1) = g(\mathbf{x}, n, f(\mathbf{x}, n))$$

where  $g : \mathbb{N}^k \times \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  and  $h : \mathbb{N}^k \rightarrow \mathbb{N}$  are partial recursive functions.

$p_f$  is defined as:

$$p_f(\mathbf{x}, n)(z_n) = \begin{cases} 1 & \text{if } z_n = \text{rec}(h, g) \\ 0 & \text{otherwise} \end{cases}$$

We see that  $h, g$  are all partial recursive functions. So we have by definition of  $p_f$  that

$$p_h(\mathbf{x})(z_0) = \begin{cases} 1 & \text{if } y = h(\mathbf{x}) \\ 0 & \text{otherwise} \end{cases}$$

$$p_g(\mathbf{x}, n, z_n)(z_{n+1}) = \begin{cases} 1 & \text{if } z_{n+1} = g(\mathbf{x}, n, f(\mathbf{x}, n)) \\ 0 & \text{otherwise} \end{cases}$$

By hypothesis we observe that  $p_g, p_h \in \mathcal{PR}$ . Now if  $n = 0$  then  $p_f(\mathbf{x}, 0) = p_h(\mathbf{x})$  and if  $n > 0$  then  $p_f(\mathbf{x}, n+1) = p_g(\mathbf{x}, n, z_n)$ . We observe that

$$\begin{aligned} ((p_g \odot (id, p_f))(\mathbf{x}, n))(z_{n+1}) &= \sum_{x_1, \dots, z_k, n, z_n \in \mathbb{N}} p_g(x_1, \dots, x_k, n, z_n)(z_{n+1}) \cdot \left( \prod_{1 \leq i \leq k+1} id(\mathbf{x}, n)(\mathbf{x}, n) \cdot f(\mathbf{x}, n)(z_n) \right) \\ &= \sum_{x_1, \dots, z_k, n, z_n \in \mathbb{N}} p_g(x_1, \dots, x_k, n, z_n)(z_{n+1}) \cdot \left( \prod_{\mathbf{x}=\mathbf{x}, n=n, z_n=f(\mathbf{x}, n)} 1 \right) \\ &= \sum_{z_{n+1}=g(x_1, \dots, x_k, n, z_n)} 1 \cdot \left( \prod_{\mathbf{x}=\mathbf{x}, n=n, z_n=f(\mathbf{x}, n)} 1 \right) \\ &= \sum_{z_{n+1}=g(\mathbf{x}, n, f(\mathbf{x}, n))} 1 \\ &= p_f(\mathbf{x}, n+1)(z_{n+1}) \end{aligned}$$

Thus  $p_f$  is in  $\mathcal{PR}$  because  $p_f = \text{rec}(p_h, p_g)$ .

- $f$  is defined by minimization so:

$$f(\mathbf{x}) = \mu y (g(\mathbf{x}, y) = 0)$$

$p_f$  is defined as:

$$p_f(\mathbf{x})(z) = \begin{cases} 1 & \text{if } z = f(\mathbf{x}) \\ 0 & \text{otherwise} \end{cases}$$

by definition of  $f(\mathbf{x})$  we have:

$$p_f(\mathbf{x})(z) = \begin{cases} 1 & \text{if } z = \mu y (g(\mathbf{x}, y) = 0) \\ 0 & \text{otherwise} \end{cases}$$

We know that  $g$  is a recursive function, so we have that:

$$p_g(\mathbf{x}, z)(k) = \begin{cases} 1 & \text{if } k = g(\mathbf{x}, z) \\ 0 & \text{otherwise} \end{cases}$$

By hypothesis  $p_g \in \mathcal{PR}$ . We observe that:

$$\begin{aligned}
\mu p_g(\mathbf{x})(z) &= p_g(\mathbf{x}, z)(0) \cdot \left( \prod_{n < z} \left( \sum_{k > 0} p_g(\mathbf{x}, n)(k) \right) \right) \\
&= p_g(\mathbf{x}, z)(0) \cdot \left( \prod_{n < z} \left( \sum_{k > 0, k = g(\mathbf{x}, n)} 1 \right) \right) \\
&= p_g(\mathbf{x}, z)(0) \cdot \left( \prod_{n < z, k > 0, k = g(\mathbf{x}, n)} 1 \right) \\
&= \begin{cases} 1 & \text{if } z \text{ is the minimal values such that } g(\mathbf{x}, z) = 0 \text{ and for all } n < z \text{ } g(\mathbf{x}, n) > 0 \\ 0 & \text{otherwise} \end{cases} \\
&= \begin{cases} 1 & \text{if } z = \mu y (g(\mathbf{x}, y) = 0) \\ 0 & \text{otherwise} \end{cases} \\
&= p_f(\mathbf{x})(z)
\end{aligned}$$

Thus  $p_f$  is in  $\mathcal{PR}$  because  $p_f = \mu p_g$

□

It is easy to show that  $\mathcal{PR}$  includes all partial recursive functions, seen as probabilistic functions: first, for every partial function  $f : \mathbb{N}^k \rightarrow \mathbb{N}$ , define  $p_f : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$  by stipulating that  $p_f(\mathbf{x})(y) = 1$  whenever  $y = f(\mathbf{x})$ , and  $p_f(\mathbf{x})(y) = 0$  otherwise; then,  $p_f \in \mathcal{PR}$  whenever  $f$  is partial recursive.

**Example 1** *The following are examples of probabilistic recursive functions:*

- The identity function  $id : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$ , defined as  $id(x)(y) = 1$ . For all  $x, y \in \mathbb{N}$  we have that

$$id(x)(y) = \begin{cases} 1 & \text{if } y = x \\ 0 & \text{otherwise} \end{cases}$$

on the other hand for every  $x, y \in \mathbb{N}$  we have as a consequence,  $id = \Pi_1^1$ , and, since the latter is a basic function (Definition 2)  $id$  is in  $\mathcal{PR}$ .

- The function  $f : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  defined by  $f(x)(x) = \frac{1}{2}$  and  $f(x)(x+1) = \frac{1}{2}$ . We define  $id : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  as follow:

$$id(x)(y) = \begin{cases} 1 & \text{if } y = x \\ 0 & \text{otherwise} \end{cases}$$

We define  $g : \mathbb{N}^3 \rightarrow \mathbb{P}_{\mathbb{N}}$  as follow:

$$g(x_1, x_2, z)(y) = \begin{cases} 1 & \text{if } y = z + 1 \\ 0 & \text{otherwise} \end{cases}$$

$g$  is in  $\mathcal{PR}$  because  $g = s \odot (\Pi_3^3)$

Finally the function  $add$  satisfies the follow equations:

$$\begin{aligned}
add(x, 0) &= h(x); \\
add(x_1, x_2 + 1) &= g(x_1, x_2, add(x_1, x_2))
\end{aligned}$$

so we proved that  $add$  is constructed by Primitive Recursion operation and it is a probabilistic recursive function. Now we observe that:

$$f = add \odot (\Pi_1^1, rand)$$

that is a function defined using the operation of General Composition (Definition 4).

- The function  $f : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  defined by

$$f(x)(y) = \begin{cases} \frac{1}{2^{y-x}} & \text{if } y > x \\ 0 & \text{otherwise} \end{cases}$$

We define  $h : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  as follow:

$$h(x) = \begin{cases} 1/2^y & \text{if } y \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

that is a probabilistic recursive function because

$$\mu \text{ rand}(x)(y) = \text{rand}(x, y)(0) \cdot \left( \prod_{y < z} \left( \sum_{k > 0} \text{rand}(x, z)(k) \right) \right)$$

thus  $\text{rand}(x, y)(0) = 1/2$  and  $\sum_{k > 0} \text{rand}(x, z)(k) = \sum_{k=1,2,\dots} \text{rand}(x, z)(k) = \text{rand}(x, z)(1) + \text{rand}(x, z)(2) + \dots = 1/2 + 0 + \dots = 1/2$  for definition of  $\text{rand}$ .  $\prod_{y < z} 1/2 = 1/2^{y-1}$ . So

$$\mu \text{ rand}(x)(y) = \begin{cases} 1/2^y & \text{if } y \geq 1 \\ 0 & \text{otherwise} \end{cases}$$

Then we observe that:

$$g(x)(y) = \text{add} \odot (\mu \text{ rand}, \text{id})(x)(y)$$

So

$$\text{add} \odot (\mu \text{ rand}, \text{id})(x)(y) = \sum_{x_1, x_2} \text{add}(x_1, x_2)(y) \cdot (\mu \text{ rand}(x)(x_1) * \text{id}(x)(x_2)) = 1/2^{y-x}$$

because the function  $\text{id}(x)(x) = 1$  and so  $x_2 = x$  and  $\mu \text{ rand}(x)(x_1) = 1/2^{x_1}$  if  $x_1 > 0$ . So  $x_1 > 0$ . Now this is true if and only if  $x_2 = x$  and  $x + x_1 = y$  and finally  $x_1 = y - x$ .

## 2.1 Probabilistic Turing Machines and Computable Functions

In this section we introduce computable functions as those probabilistic functions which can be computed by Probabilistic Turing Machines. As previously mentioned, probabilistic computation devices have received a wide interest in computer science already in the fifties [5] and early sixties [14]. A natural question which arose was then to see what happened if random elements were allowed in a Turing machine. This question led to several formalizations of probabilistic Turing machines (PTMs in the following) [5, 16] — which, essentially, are Turing machines which have the ability to flip coins in order to make random decisions — and to several results concerning the computational complexity of problems when solved by PTMs [6].

Following [6], a Probabilistic Turing Machine (PTM)  $M$  can be seen as a Turing Machine with two transition functions  $\delta_0, \delta_1$ . At each computation step, either  $\delta_0$  or  $\delta_1$  can be applied, each with probability  $1/2$ . Then, in a way analogous to the deterministic case, we can define a notion of a (initial, final) configuration for a PTM  $M$ . In the following,  $\Sigma_b$  denotes the set of possible symbols on the tape, including a blank symbol;  $Q$  denotes the set of states;  $Q_f \subseteq Q$  denotes the set of final states and  $q_s \in Q$  denotes the initial state.

**Definition 9 (Probabilistic Turing Machine)** A Probabilistic Turing Machine (PTM) is a Turing machine endowed with two transition functions  $\delta_0, \delta_1$ . At each computation step the transition function  $\delta_0$  can be applied with probability  $1/2$  and the transition  $\delta_1$  can be applied with probability  $1/2$ .

**Definition 10 (Configuration of a PTM)** Let  $M$  be a PTM. We define a PTM configuration as a 4-tuple  $\langle s, a, t, q \rangle \in \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q$  such that:

- The first component,  $s \in \Sigma_b^*$ , is the portion of the tape lying on the left of the head.



- The second component,  $a \in \Sigma_b$ , is the symbol the head is reading.
- The third component,  $t \in \Sigma_b^*$ , is the portion of the tape lying on the right of the head.
- The fourth component,  $q \in Q$  is the current state.

Moreover we define the set of all configurations as  $\mathcal{C}_M = \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q$ .

**Definition 11 (Initial and Final Configurations of a PTM)** *Let  $M$  be a PTM. We define the initial configuration of  $M$  for the string  $s$  as the configuration in the form  $\langle \varepsilon, a, v, q_s \rangle \in \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q$  such that  $s = a \cdot v$  and the fourth component,  $q_s \in Q$ , is the initial state. We denote it with  $\mathcal{IN}_M^s$ . Similarly, we define a final configuration of  $M$  for  $s$  as a configuration  $\langle s, \varepsilon, q_f \rangle \in \Sigma_b^* \times \Sigma_b \times \Sigma_b^* \times Q_f$ . The set of all such final configurations for a PTM  $M$  is denoted by  $\mathcal{FC}_M^s$ .*

For a function  $T : \mathbb{N} \rightarrow \mathbb{N}$ , we say that a PTM  $M$  runs in time bounded by  $T$  if for any input  $x$ ,  $M$  halts on input  $x$  within  $T(|x|)$  steps independently of the random choices it makes. Thus,  $M$  works in polynomial time if it runs in time bounded by  $P$ , where  $P$  is any polynomial.

Intuitively, the function computed by a PTM  $M$  associates to each input  $s$ , a (pseudo)-distribution which indicates the probability of reaching a final configuration of  $M$  from  $\mathcal{IN}_M^s$ . It is worth noticing that, differently from the deterministic case, since in a PTM the same configuration can be obtained by different computations, the probability of reaching a given final configuration is the *sum* of the probabilities of reaching the configuration along all computation paths, of which there can be (even infinitely) many. It is thus convenient to define the function computed by a PTM through a fixpoint construction, as follows. First, we can define a partial order on the string distributions as follows.

**Definition 12** *A string distribution on  $\Sigma^*$  is a function  $\mathcal{D} : \Sigma^* \rightarrow \mathbb{R}_{[0,1]}$  such that  $\sum_{s \in \Sigma^*} \mathcal{D}(s) \leq 1$ .  $\mathbb{P}_{\Sigma^*}$  denotes the set of all string distributions on  $\Sigma^*$ .*

Next we can define a partial order on string distributions by a pointwise extension of the usual order on  $\mathbb{R}$ :

**Definition 13** *The relation  $\sqsubseteq_{\mathbb{P}_{\Sigma^*}} \subseteq \mathbb{P}_{\Sigma^*} \times \mathbb{P}_{\Sigma^*}$  is defined by stipulating that  $A \sqsubseteq_{\mathbb{P}_{\Sigma^*}} B$  if and only if, for all  $s \in \Sigma^*$ ,  $A(s) \leq B(s)$ .*

The proof of the following is immediate.

**Proposition 2** *The structure  $(\mathbb{P}_{\Sigma^*}, \sqsubseteq_{\mathbb{P}_{\Sigma^*}})$  is a POSET.*

Now we can define the domain  $\mathcal{CEV}$  of those functions computed by a PTM  $M$  from a given configuration<sup>1</sup>. This set is defined as follows and will be used as the domain of the functional whose least fixpoint gives the function computed by a PTM.

**Definition 14** *The set  $\mathcal{CEV}$  is defined as  $\{f | f : \mathcal{C}_M \rightarrow \mathbb{P}_{\Sigma^*}\}$*

Inheriting the structure on  $\mathbb{P}_{\Sigma^*}$  we can define a partial order on  $\mathcal{CEV}$  as follows.

**Definition 15** *The relation  $\sqsubseteq_{\mathcal{CEV}} \subseteq \mathcal{CEV} \times \mathcal{CEV}$  is defined for  $A, B \in \mathcal{CEV}$   $A \sqsubseteq_{\mathcal{CEV}} B$  if and only if, for all  $c \in \mathcal{C}_M$ ,  $A(c) \sqsubseteq_{\mathbb{P}_{\Sigma^*}} B(c)$*

Also the proof of the following is immediate.

**Proposition 3** *The structure  $(\mathcal{CEV}, \sqsubseteq_{\mathcal{CEV}})$  is a POSET.*

Given a POSET, the notions of least upper bound, denoted by  $\sqcup$ , and of an ascending chain are defined as usual. Next, the bottom elements are defined as follows.

**Lemma 1** *Let  $d_\perp : \Sigma^* \rightarrow \mathbb{R}_{[0,1]}$  be defined by stipulating that  $d_\perp(s) = 0$  for all  $s \in \Sigma^*$ . Then,  $d_\perp$  is the bottom element of the poset  $(\mathbb{P}_{\Sigma^*}, \sqsubseteq_{\mathbb{P}_{\Sigma^*}})$ .*

---

<sup>1</sup>Of course  $\mathcal{CEV}$  is a proper superset of the functions computed by PTMs.

**Lemma 2** Let  $b_\perp : \mathcal{C}_M \rightarrow \mathbb{P}_{\Sigma^*}$  be defined by stipulating that  $b_\perp(c) = d_\perp$  for all  $c \in \mathcal{C}_M$ . Then,  $b_\perp$  is the bottom element of the poset  $(\mathcal{CEV}, \sqsubseteq_{\mathcal{CEV}})$ .

Now, it is time prove that the posets at hand are also  $\omega$ -complete:

**Proposition 4** The POSET  $(\mathbb{P}_{\Sigma^*}, \sqsubseteq_{\mathbb{P}_{\Sigma^*}})$  is a  $\omega$ CPO.

**Proof.** We need to prove that for each chain

$$c_1 \sqsubseteq_{\mathbb{P}_{\Sigma^*}} c_2 \sqsubseteq_{\mathbb{P}_{\Sigma^*}} c_3 \dots$$

the least upper bound  $\bigsqcup_i c_i$  exists. First note that since  $\sum_{s \in \Sigma^*} c_i(s) \leq 1$ , from definition of  $\sqsubseteq_{\mathbb{P}_{\Sigma^*}}$  it follows that, for each  $s \in \Sigma^*$ ,  $c_1(s) \leq c_2(s) \leq \dots \leq 1$  holds. This implies that, for each  $s \in \Sigma^*$ , the limit  $\lim_{i \rightarrow \infty} c_i(s)$  exists. Hence, defining  $c_{LIM}$  as the distribution such that  $c_{LIM}(s) = \lim_{i \rightarrow \infty} c_i(s)$ , we have that  $c_{LIM} = \bigsqcup_i c_i$ . Indeed,  $c_{LIM} \sqsupseteq_{\mathbb{P}_{\Sigma^*}} c_i$ , and any upper bounds of the family  $\{c_i\}_{i \in \mathbb{N}}$  is clearly greater or equal to  $c_{LIM}$ .  $\square$

**Proposition 5** The POSET  $(\mathcal{CEV}, \sqsubseteq_{\mathcal{CEV}})$  is a  $\omega$ CPO.

**Proof.** Analogous to the previous one.  $\square$

We can now define a functional  $F_M$  on  $\mathcal{CEV}$  which will be used to define the function computed by  $M$  via a fixpoint construction. Intuitively, the application of the functional  $F_M$  describes *one* computation step. Formally:

**Definition 16** Given a PTM  $M$ , we define a functional  $F_M : \mathcal{CEV} \rightarrow \mathcal{CEV}$  as:

$$F_M(f)(C) = \begin{cases} \{s^1\} & \text{if } C \in \mathcal{FC}_M^s; \\ \frac{1}{2}f(\delta_0(C)) + \frac{1}{2}f(\delta_1(C)) & \text{otherwise.} \end{cases}$$

The following proposition is needed in order to apply the usual fix point result.

**Proposition 6** The functional  $F_M$  is continuous.

**Proof.** We prove that

$$F_M(\bigsqcup_{i \in \mathbb{N}} f_i) = \bigsqcup_{i \in \mathbb{N}} (F_M(f_i)),$$

or, saying another way, that for every configuration  $C$ ,

$$F_M(\bigsqcup_{i \in \mathbb{N}} f_i)(C) = \bigsqcup_{i \in \mathbb{N}} (F_M(f_i))(C).$$

Now, notice that for every  $C$ ,

$$F_M(\bigsqcup_{i \in \mathbb{N}} f_i)(C) = \begin{cases} \{s^1\} & \text{if } C \in \mathcal{FC}_M^s \\ \frac{1}{2}((\bigsqcup_{i \in \mathbb{N}} f_i)(C_1)) + \frac{1}{2}((\bigsqcup_{i \in \mathbb{N}} f_i)(C_2)) & \text{if } C \rightarrow C_1, C_2 \end{cases}$$

and, similarly, that:

$$\bigsqcup_{i \in \mathbb{N}} (F_M(f_i))(C) = \bigsqcup_{i \in \mathbb{N}} \begin{cases} \{s^1\} & \text{if } C \in \mathcal{FC}_M^s \\ \frac{1}{2}f_i(C_1) + \frac{1}{2}f_i(C_2) & \text{if } C \rightarrow C_1, C_2 \end{cases}$$

Now, given any  $C$ , we distinguish two cases:

- If  $C \in \mathcal{FC}_M^s$ , then

$$F_M(\bigsqcup_{i \in \mathbb{N}} f_i)(C) = \{s^1\} = \bigsqcup_{i \in \mathbb{N}} \{s^1\} = \bigsqcup_{i \in \mathbb{N}} (F_M(f_i))(C).$$

- If  $C \rightarrow C_1, C_2$ , then

$$\begin{aligned}
F_M(\bigsqcup_{i \in \mathbb{N}} f_i)(C) &= \frac{1}{2}((\bigsqcup_{i \in \mathbb{N}} f_i)(C_1)) + \frac{1}{2}((\bigsqcup_{i \in \mathbb{N}} f_i)(C_2)) \\
&= \frac{1}{2}(\bigsqcup_{i \in \mathbb{N}} f_i(C_1)) + \frac{1}{2}(\bigsqcup_{i \in \mathbb{N}} f_i(C_2)) \\
&= \bigsqcup_{i \in \mathbb{N}} \frac{1}{2}f_i(C_1) + \bigsqcup_{i \in \mathbb{N}} \frac{1}{2}f_i(C_2) = \bigsqcup_{i \in \mathbb{N}} (\frac{1}{2}f_i(C_1) + \frac{1}{2}f_i(C_2)) \\
&= \bigsqcup_{i \in \mathbb{N}} (F_M(f_i))(C).
\end{aligned}$$

This concludes the proof.  $\square$

**Theorem 1** *The functional defined in 16 has a least fix point which is equal to  $\bigsqcup_{n \geq 0} F_M^n(b_\perp)$ .*

**Proof.** Immediate from the well-known fix point theorem for continuous maps on a  $\omega$ CPO.  $\square$

Such a least fixpoint is, once composed with a function returning  $\mathcal{LN}_M^s$  from  $s$ , the *function computed by the machine  $M$* , which is denoted as  $\mathcal{IO}_M : \Sigma^* \rightarrow \mathbb{P}_{\Sigma^*}$ . The set of those functions which can be computed by any PTMs is denoted as  $\mathcal{PC}$ .

The notion of a computable probabilistic function subsumes other key notions in probabilistic and real-number computation. As an example, *computable distributions* can be characterized as those distributions on  $\Sigma^*$  which can be obtained as the result of a function in  $\mathcal{PC}$  on a *fixed* input. Analogously, *computable real numbers* from the unit interval  $[0, 1]$  can be seen as those elements of  $\mathbb{R}$  in the form  $f(0)(0)$  for a computable function  $f \in \mathcal{PC}$ .

## 2.2 Probabilistic Recursive Functions equals Functions computed by Probabilistic Turing Machines

In this section we prove that probabilistic *recursive* functions are the same as probabilistic *computable* functions, modulo an appropriate bijection between strings and natural numbers which we denote (as its inverse) with  $(\cdot)$ .

In order to prove the equivalence result we first need to show that a probabilistic recursive function can be computed by a PTM. This result is not difficult and, analogously to the deterministic case, is proved by exhibiting PTMs which simulate the basic probabilistic recursive functions and by showing that  $\mathcal{PC}$  is closed by composition, primitive recursion, and minimization. This is done by the following Lemmata.

**Lemma 3 (Basic Functions are Computable)** *All Basic Probabilistic Functions are Computable.*

**Proof.** For every basic function from Definition 2, we can construct a Probabilistic Turing Machine that computes it quite easily. More specifically, the proof is immediate for functions,  $z, s, \Pi$ , by observing that they are deterministic, thus the usual Turing machine for them (seen as a PTM). As for the function *rand* it can be simulated by a PTM  $M$  which operates as follows:

1.  $M$  deletes all the input written on the tape;
2.  $M$  writes  $\bar{1}$  or  $\bar{0}$  on the tape, both with probability  $1/2$ , and then halts.

This concludes the proof.  $\square$

The composition of two computable probabilistic functions is itself computable:

**Lemma 4 (Generalized Composition and Computability)** *Given Turing-Computable  $f : \mathbb{N}^n \rightarrow \mathbb{P}_{\mathbb{N}}$ , and  $g_1 : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}, \dots, g_n : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$  the function  $f \odot (g_1, \dots, g_n) : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$  is itself Turing-Computable*

**Proof.** We give an informal proof. We define PTM, said  $M$  working on  $n + 2$  tapes. (We know that PTMs with  $m > 1$  tapes compute the same class of functions of PTMs with a single tape.) The first tape is the input tape, on the next  $n + 1$  tapes  $M$  computes  $g_1, \dots, g_n$ , while on the last tape,  $M$  computes the function  $f$  on the results of  $g_1, \dots, g_n$ . The machine  $M$  operates as follows:

1. it copies the input from the first to the next  $n$  tapes;
2. in the  $i + 1$ -th tape, the machine  $M$  computes the respective function  $g_i$ , where  $1 \leq i \leq n$ ; this can of course be done, because, by induction, the  $g_i$  are computable;
3. it copies the  $n$  outputs in the  $n$  tapes numbered  $2, \dots, n + 1$  to the last tape;
4. computes the function  $f$  on the last tape and return the result  $z$ .

This concludes the proof.  $\square$

**Lemma 5 (Primitive Recursion and Turing-Computability)** *Given Turing-Computable  $g : \mathbb{N}^{k+2} \rightarrow \mathbb{P}_{\mathbb{N}}$  and  $f : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$  the function  $\text{rec}(f, g) : \mathbb{N}^{k+1} \rightarrow \mathbb{P}_{\mathbb{N}}$  is itself Turing-Computable.*

**Proof.** We give an informal proof. We define PTM, said  $M$  working on 5 tapes. The first tape is the input tape, on the next tape  $M$  computes the count down of our  $k + 1^{\text{th}}$  variable, on the third tape  $M$  computes  $g$ , on the fourth tape  $M$  computes the function  $f$ , and in the last saves the result. The machine  $M$  operates as follows:

1. it copies in the second tape the  $k + 1^{\text{th}}$  element of the input, and then it copies on the fourth tape the first  $k$  elements of the input;
2. it computes  $f$  and saves the result on the last tape;
3. it verifies if the second tape is 0. In this case  $M$  stops and the last tape contains the result, otherwise it copies the first  $k$  elements of the input from the first tape in the third tape and then it copies the result present in the last tape on the third tape;
4.  $M$  decrements of the value on the second tape;
5. it computes  $g$  on the third tape and save the result on the last tape;
6. it returns to the step 3.

This concludes the proof.  $\square$

**Lemma 6 (Minimization and Turing-Computability)** *Given Turing-Computable  $f : \mathbb{N}^{k+1} \rightarrow \mathbb{P}_{\mathbb{N}}$ , the function  $\mu f : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$  is itself Turing-Computable.*

**Proof.** We give an intuitive proof. We take a PTM, said  $M$  with 4 tapes. The first tape is the input tape, on the next tape  $M$  saves one element that we name  $y$ , on the third tape it computes the function  $f$  and in the last tape it saves the result. The machine  $M$  operates as follows:

1. it writes in the second tape 0 and it copies on the third tape the input and the value  $y$  (present in the second tape);
2. it computes on the third tape the function  $f$  and saves the result on the last tape;
3. it verifies if the last tape contains the value 0. In this case it saves on the last tape the element in the second tape and it stops, otherwise it increases  $y$ ;
4. it copies in the third tape the input and  $y$ ;
5. it returns to the step 3.

This concludes the proof.  $\square$

Hence we can prove the following theorem, showing that Probabilistic Recursive Functions are computable by a Probabilistic Turing Machine.

**Theorem 2**  $\mathcal{PR} \subseteq \mathcal{PC}$

**Proof.** Immediate from Lemmata 3, 4, 5 and 6.  $\square$

The most difficult part of the equivalence proof consists in proving that each probabilistic computable function is actually *recursive*. Analogously to the classic case, a good strategy consists in representing configurations as natural numbers, then encoding the transition functions of the machine at hand, call it  $M$ , as a (recursive) function on  $\mathbb{N}$ . In the classic case the proof proceeds by making essential use of the minimization operator by which one determines the *number* of transition steps of  $M$  necessary to reach a final configuration, if such number exists. This number can then be fed into another function which simulates  $M$  (on an input) a given number of steps, and which is primitive recursive. In our case, this strategy does not work: the number of computation steps can be infinite, even when the probability of converging is 1. Given our definition of minimization which involves distributions, this is delicate, since we have to define a suitable function on the PTM computation tree to be minimized.

In order to do adapt the classic proof, we need to formalize the notion of a *computation tree* which represents all computation paths corresponding to a given input string  $x$ . We define such a tree as follows. Each node is labelled by a configuration of the machine and each edge represents a computation step. The root is labelled with  $\mathcal{N}_M^x$  and each node labelled with  $C$  has either no child (if  $C$  is final) or 2 children (otherwise), labelled with  $\delta_0(C)$  and  $\delta_1(C)$ . Please notice that the same configuration may be duplicated across a single level of the tree as well as appear at different levels of the tree; nevertheless we represent each such appearance by a separate node.

We can naturally associate a probability with each node, corresponding to the probability that the node is reached in the computation: it is  $\frac{1}{2^n}$ , where  $n$  is the height of the node. The probability of a particular *final* configuration is the sum of the probabilities of all leaves labelled with that configuration. We also enumerate nodes in the tree, top-down and from left to right, by using binary strings in the following way: the root has associated the number  $\varepsilon$ . Then if  $b$  is the binary string representing the node  $N$ , the left child of  $N$  has associated the string  $b \cdot 0$  while the right child has the number  $b \cdot 1$ . Note that from this definition it follows that each binary number associated to a node  $N$  indicates a path in the tree from the root to  $N$ . The computation tree for  $x$  will be denoted as  $CT_M(x)$ .

We give now a more explicit description of the constructions described above. First we need to encode the rational numbers  $\mathbb{Q}$  into  $\mathbb{N}$ . Let  $pair : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  be any recursive bijection between pairs of natural numbers and natural numbers such that  $pair$  and its inverse are both computable. Let then  $enc$  be just  $p_{pair}$ , i.e. the function  $enc : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  defined as follows

$$enc(a, b)(q) = \begin{cases} 1 & \text{if } q = pair(a, b) \\ 0 & \text{otherwise} \end{cases}$$

The function  $enc$  allows to represent positive rational numbers as pairs of natural numbers in the obvious way and is recursive.

It is now time to define a few notions on computation trees

**Definition 17 (Computation Trees and String Probabilities)** *The function  $PT_M : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{Q}$  is defined by stipulating that  $PT_M(x, y)$  is the probability of observing the string  $\bar{y}$  in the tree  $CT_M(x)$ , namely  $\frac{1}{2^{|\bar{y}|}}$ .*

Of course,  $PT_M$  is partial recursive, thus  $p_{PT_M}$  is probabilistic recursive. Since the same configuration  $C$  can label more than one node in a computation tree  $CT_M(x)$ ,  $PT_M$  does not indicate the probability of reaching  $C$ , even when  $C$  is the label of the node corresponding to the second argument. Such a probability can be obtained by summing the probability of all nodes labelled with the same configuration at hand:

**Definition 18 (Configuration Probability)** *Suppose given a PTM  $M$ . If  $x \in \mathbb{N}$  and  $z \in \mathcal{C}_M$ , the subset  $CC_M(x, z)$  of  $\mathbb{N}$  contains precisely the indices of nodes of  $CT_M(x)$  which are labelled by  $z$ . The function  $PC_M : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{Q}$  is defined as follows:*

$$PC_M(x, z) = \sum_{y \in CC_M(x, z)} PT_M(x, y)$$

Contrary to  $PT_M$ , there is nothing guaranteeing that  $PC_M$  is indeed computable. In the following, however, what we will do is precisely showing that this is the case.

In Figure 1 we show an example of computation tree  $CT_M(x)$  for an hypothetical PTM  $M$  and an input  $x$ . The leaves, depicted as red nodes, represent the final configurations of the computation. So, for example,  $PC_M(x, C) = 1$ , while  $PC_M(x, E) = \frac{3}{4}$ . Indeed, notice that there are three nodes in the tree which are labelled with  $E$ , namely those corresponding to the binary strings 00, 01, and 10. As we already mentioned, our proof separates the classic part of the

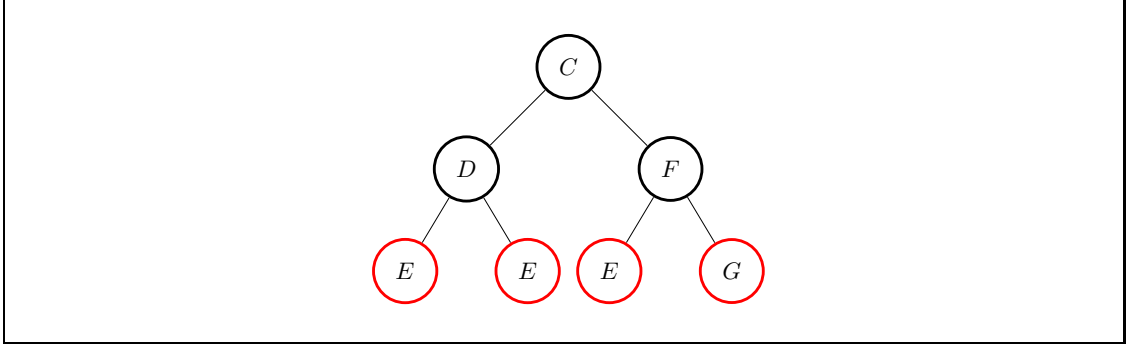


Figure 1: An Example of a Computation Tree

computation performed by the underlying PTM, which essentially computes the configurations reached by the machine in different paths, from the probabilistic part, which instead computes the probability values associated to each computation by using minimization. These two tasks are realized by two suitable probabilistic recursive functions, which are then composed to obtain the function computed by the underlying PTM. We start with the probabilistic part, which is more complicated.

We need to define a function, which returns the *conditional* probability of terminating at the node corresponding to the string  $\bar{y}$  in the tree  $CT_M(x)$ , given that all the nodes  $\bar{z}$  where  $z < y$  are labelled with non-final configurations. This is captured by the following definition:

**Definition 19** *Given a PTM  $M$ , we define  $PT_M^0 : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{Q}$  and  $PT_M^1 : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{Q}$  as follows:*

$$PT_M^1(x, y) = \begin{cases} 1 & \text{if } y \text{ is not a leaf of } CT_M(x); \\ 1 - PT_M^0(x, y) & \text{otherwise;} \end{cases}$$

$$PT_M^0(x, y) = \begin{cases} 0 & \text{if } y \text{ is not a leaf of } CT_M(x); \\ \frac{PT_M(x, y)}{\prod_{k < y} PT_M^1(x, k)} & \text{otherwise;} \end{cases}$$

Note that, according to previous definition,  $PT_M^1(x, y)$  is the probability of *not* terminating the computation in the node  $y$ , while  $PT_M^0(x, y)$  represents the probability of terminating the computation in the node  $y$ , both *knowing* that the computation has not terminated in any node  $k$  preceding  $y$ .

**Proposition 7** *The functions  $PT_M^0 : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{Q}$  and  $PT_M^1 : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{Q}$  are partial recursive.*

**Proof.** Please observe that  $PT_M$  is partial recursive and that the definitions above are mutually recursive, but the underlying order is well-founded. Both functions are thus intuitively computable, thus partial recursive by the Church-Turing thesis.  $\square$   $\square$

The reason why the two functions above are useful is because they associate the distribution  $\{0^{PT_M^1(x, y)}, 1^{PT_M^0(x, y)}\}$  to each pair of natural numbers  $(x, y)$ . In Figure 2, we give the quantities

we have just defined for the tree from Figure 1. Each internal node is associated with the same distribution  $\{0^0, 1^1\}$ . Only the leaves are associated with nontrivial distributions. As an example, the distribution associated to the node 10 is  $\{0^{1/2}, 1^{1/2}\}$ , because we have that

$$\begin{aligned} PT_M^0(x, \overline{10}) &= \frac{PT_M(x, \overline{10})}{\prod_{k < \overline{10}} PT_M^1(x, k)} \\ &= \frac{1}{4 \cdot PT_M^1(x, \overline{01}) \cdot PT_M^1(x, \overline{00}) \cdot PT_M^1(x, \overline{1}) \cdot PT_M^1(x, \overline{0}) \cdot PT_M^1(x, \overline{\varepsilon})} \\ &= \frac{1}{4 \cdot PT_M^1(x, \overline{01}) \cdot PT_M^1(x, \overline{00})}. \end{aligned}$$

As it can be easily verified,  $PT_M^1(x, \overline{00}) = \frac{3}{4}$ , while  $PT_M^1(x, \overline{01}) = \frac{2}{3}$ . Thus,  $PT_M^0(x, \overline{10}) = \frac{1}{2}$ .

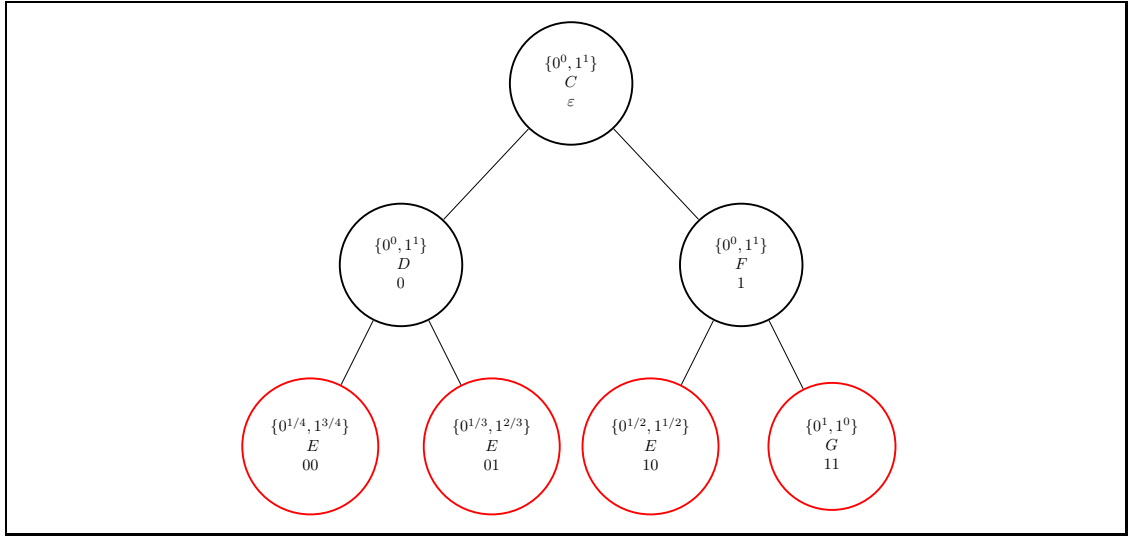


Figure 2: The Conditional Probabilities for the Computation Tree from Figure 1

We now need to go further, and prove that the probabilistic function returning, on input  $(x, y)$ , the distribution  $\{0^{PT_M^1(x, y)}, 1^{PT_M^0(x, y)}\}$  is recursive. This is captured by the following definition:

**Definition 20** Given a PTM  $M$ , the function  $PTC_M : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  is defined as follows

$$PTC_M(x, y)(z) = \begin{cases} PT_M^0(x, y) & \text{if } z = 0; \\ PT_M^1(x, y) & \text{if } z = 1; \\ 0 & \text{otherwise} \end{cases}$$

The function  $PTC_M$  is really the core of our encoding. On the one hand, we will show that it is indeed recursive. On the other, minimizing it is going to provide us exactly with the function we need to reach our final goal, namely proving that the probabilistic function computed by  $M$  is itself recursive. But how should we proceed if we want to prove  $PTC_M$  to be recursive? The idea is to compose  $p_{PT_M^1}$  with a function that turns its input into the probability of returning 1. This is precisely what the following function does:

**Definition 21** The function  $I2P : \mathbb{Q} \rightarrow \mathbb{P}_{\mathbb{N}}$  is defined as follows

$$I2P(x)(y) = \begin{cases} x & \text{if } (0 \leq x \leq 1) \wedge (y = 1) \\ 1 - x & \text{if } (0 \leq x \leq 1) \wedge (y = 0) \\ 0 & \text{otherwise} \end{cases}$$

Please observe how the input to  $I2P$  is the set of rational numbers, as usual encoded by pairs of natural numbers. Previous definitions allow us to treat (rational numbers representing) probabilities in our algebra of functions. Indeed:

**Proposition 8** *The probabilistic function  $I2P$  is recursive.*

**Proof.** We first observe that  $h : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  defined as

$$h(x)(y) = 1/2^{y+1}$$

is a probabilistic recursive function, because  $h = \mu \text{ (rand } \odot \Pi_1^2)$ . Next we observe that every  $q \in \mathbb{Q} \cap [0, 1]$  can be represented in binary notation as:

$$q = \sum_{i \in \mathbb{N}} \frac{c_i^q}{1/2^{i+1}}$$

where  $c_i^q \in \{0, 1\}$  (i.e.,  $c_i^q$  is the  $i$ -th element of the binary representation of  $q$ ). Moreover, a function computing such a  $c_i^q$  from  $q$  and  $i$  is partial recursive. Hence we can define  $b : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  as follows

$$b(q, i)(y) = \begin{cases} 1 & \text{if } y = c_i^q \\ 0 & \text{otherwise} \end{cases}$$

and conclude that  $b$  is indeed a probabilistic recursive function (because  $\mathcal{PR}$  includes all the partial recursive functions, seen as probabilistic functions). Observe that:

$$b(q, i)(y) = \begin{cases} c_i^q & \text{if } y = 1 \\ 1 - c_i^q & \text{if } y = 0 \end{cases}$$

From the definition of composition, it follows that

$$\begin{aligned} (b \odot (id, h))(q)(y) &= \sum_{x_1, x_2} b(x_1, x_2)(y) \cdot id(q)(x_1) \cdot h(q)(x_2) \\ &= \sum_{x_2} b(q, x_2)(y) \cdot h(q)(x_2) = \sum_{x_2} b(q, x_2)(y) \cdot \frac{1}{2^{x_2+1}} \\ &= \begin{cases} \sum_{x_2} \frac{c_{x_2}^q}{2^{x_2+1}} & \text{if } y = 1 \\ \sum_{x_2} \frac{1 - c_{x_2}^q}{2^{x_2+1}} & \text{if } y = 0 \end{cases} = \begin{cases} q & \text{if } y = 1 \\ 1 - q & \text{if } y = 0 \end{cases} \end{aligned}$$

This shows that

$$I2P = b \odot (id, h),$$

and hence that  $I2P$  is probabilistic recursive. □

The following is an easy corollary of what we have obtained so far:

**Proposition 9** *The probabilistic function  $PTC_M$  is recursive.*

**Proof.** Just observe that  $PTC_M = I2P \odot p_{PT_M^1}$ . □

The probabilistic recursive function obtained as the minimization of  $PTC_M$  allows to compute a probabilistic function that, given  $x$ , returns  $y$  with probability  $PT_M(x, y)$  if  $y$  is a leaf (and otherwise the probability is just 0).

**Definition 22** *The function  $\mathcal{CF}_M : \mathbb{N} \rightarrow \mathbb{P}_{\mathbb{N}}$  is defined as follows*

$$\mathcal{CF}_M(x)(y) = \begin{cases} PT_M(x, y) & \text{if } y \text{ corresponds to a leaf} \\ 0 & \text{otherwise.} \end{cases}$$

**Proposition 10** *The probabilistic function  $\mathcal{CF}_M$  is recursive.*



**Proof.** The probabilistic function  $\mathcal{CF}_M$  is just the function obtained by minimizing  $PTC_M$ , which we already know to be recursive. Indeed, if  $z$  corresponds to a leaf, then:

$$\begin{aligned}
(\mu PTC_M)(x)(z) &= PTC_M(x, z)(0) \cdot \prod_{y < z} \sum_{k > 0} PTC_M(x, y)(k) \\
&= PTC_M(x, z)(0) \cdot \prod_{y < z} PTC_M(x, y)(1) \\
&= PT_M^0(x, z) \cdot \prod_{y < z} PT_M^1(x, y) \\
&= \frac{PT_M(x, z)}{\prod_{y < z} PT_M^1(x, y)} \cdot \prod_{y < z} PT_M^1(x, y) = PT_M(x, z).
\end{aligned}$$

If, however,  $z$  does not correspond to a leaf, then:

$$\begin{aligned}
(\mu PTC_M)(x)(z) &= PTC_M(x, z)(0) \cdot \prod_{y < z} \sum_{k > 0} PTC_M(x, y)(k) \\
&= PT_M^0(x, z)(0) \cdot \prod_{y < z} \sum_{k > 0} PTC_M(x, y)(k) = 0.
\end{aligned}$$

This concludes the proof. □

We are almost ready to wrap up our result, but before proceeding further, we need to define the function  $SP_M : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$  that, given in input a pair  $(x, y)$  returns the (encoding) of the string found in the configuration labeling the node  $y$  in  $CT_M(x)$ . We can now prove the desired result:

**Theorem 3**  $\mathcal{PC} \subseteq \mathcal{PR}$ .

**Proof.** It suffices to note that, given any PTM  $M$ , the function computed by  $M$  is nothing more than

$$p_{SP_M} \odot (id, \mathcal{CF}_M).$$

Indeed, one can easily realize that a way to simulate  $M$  indeed consists in generating from  $x$ , all strings corresponding to the leaves of  $CT_M(x)$ , each with an appropriate probability. This is indeed what  $\mathcal{CF}_M$  does. What remains to be done is simulating  $p_{SP_M}$  along paths leading to final configurations, which is what  $SP_M$  does. □

We are finally ready to prove the main result of this Section:

**Corollary 1**  $\mathcal{PR} = \mathcal{PC}$

**Proof.** Immediate from Theorem 3, observing that  $\mathcal{PR} \subseteq \mathcal{PC}$  (this implication is easy to prove). □

The way we prove Corollary 1 implies that we cannot deduce Kleene's Normal Form Theorem from it: minimization has been used many times, some of them “deep inside” the construction. A way to recover Kleene's Theorem consists in replacing minimization with a more powerful operator, essentially corresponding to computing the fixpoint of a given function .

### 3 Characterizing Probabilistic Complexity by Tiering

In this section we provide a characterization of the probabilistic functions which can be computed in polynomial time by an algebra of functions acting on word algebras. More precisely, we define a type system inspired by Leivant's notion of tiering [12], which permits to rule out functions having a too-high complexity, thus allowing to isolate the class of *predicative probabilistic functions*. Our

main result in this section is that the class  $\mathcal{PPC}$  of probabilistic functions which can be computed by a PTM in *polynomial* time equals to the class of predicative probabilistic functions.

The constructions from Section 2 can be easily generalized to a function algebra on strings in a given alphabet  $\Sigma$ , which themselves can be seen a *word algebra*  $\mathbb{W}$ . In the following we provide the details of such a generalization.

**Definition 23 (String Distribution)** A pseudodistribution on  $\mathbb{W}$  is a function  $D : \mathbb{W} \rightarrow \mathbb{R}_{[0,1]}$  such that  $\sum_{w \in \mathbb{W}} D(w) = 1$ . The set  $\mathbb{P}_{\mathbb{W}}$  is defined as the set of all distribution on  $\mathbb{W}$ .

The functions in our algebra have domain  $\mathbb{W}^k$  and codomain  $\mathbb{P}_{\mathbb{W}}$ . The idea, as usual, is that  $f(x)(y) = r$  means that  $y$  is the output obtained for the input  $x$  with probability  $r$ . Base functions include a function computing the empty string, denoted  $\varepsilon$ , and concatenation with any character  $a \in \Sigma$ , denoted by  $c_a$ . Formally we define these functions as follows:

$$\begin{aligned} \varepsilon(v)(w) &= \begin{cases} 1 & \text{if } w = \varepsilon; \\ 0 & \text{otherwise.} \end{cases} \\ c_a(v)(w) &= \begin{cases} 1 & \text{if } w = a \cdot v; \\ 0 & \text{otherwise.} \end{cases} \end{aligned}$$

Note that, for every  $v \in \mathbb{W}$ , the length of the word obtained after the application of one of the constructors  $c_a$  is  $|v| + 1$  with probability 1. Projections remain available in the usual form. Indeed the function  $\Pi_m^n : \mathbb{W}^n \rightarrow \mathbb{P}_{\mathbb{W}}$  is defined as follows:

$$\Pi_m^n(\mathbf{v})(w) = \begin{cases} 1 & \text{if } w = v_m; \\ 0 & \text{otherwise.} \end{cases}$$

The only truly random functions in our algebra are probabilistic functions in the form  $r_a : \mathbb{W} \rightarrow \mathbb{P}_{\mathbb{W}}$ , which concatenates  $\Sigma$  to the input string (with probability  $\frac{1}{2}$ , or leave it unchanged (with probability  $\frac{1}{2}$ ). Formally,

$$r_a(v)(w) = \begin{cases} 1/2 & \text{if } w = a \cdot v; \\ 1/2 & \text{if } w = v; \\ 0 & \text{otherwise.} \end{cases}$$

Next we recall the concept of composition and recurrence introduced in Definition 4 and Definition 5 and we instantiate them to the case of our algebra. Generalized Composition of functions  $f : \mathbb{W}^n \rightarrow \mathbb{P}_{\mathbb{W}}$ ,  $g_1 : \mathbb{W}^k \rightarrow \mathbb{P}_{\mathbb{W}}, \dots, g_n : \mathbb{W}^k \rightarrow \mathbb{P}_{\mathbb{W}}$  as the function  $f \odot (g_1, \dots, g_n) : \mathbb{W}^k \rightarrow \mathbb{P}_{\mathbb{W}}$  defined as:

$$((f \odot (g_1, \dots, g_n))(\mathbf{v}))(w) = \sum_{x_1, \dots, x_n \in \mathbb{W}} \left( f(x_1, \dots, x_n)(w) \cdot \prod_{1 \leq i \leq n} g_i(\mathbf{v})(x_i) \right).$$

Recurrence over  $\mathbb{W}$  takes the form:

$$\begin{aligned} f(\varepsilon, \mathbf{v}) &= g_\varepsilon(\mathbf{v}) \\ f(a \cdot w, \mathbf{v}) &= g_a(f(w, \mathbf{v}), w, \mathbf{v}) \quad \forall a \in \Sigma \end{aligned}$$

where  $f : \mathbb{W} \times \mathbb{W}^m \rightarrow \mathbb{P}_{\mathbb{W}}$ , and  $g_a : \mathbb{W} \times \mathbb{W} \times \mathbb{W}^m \rightarrow \mathbb{P}_{\mathbb{W}}$  for every  $a \in \Sigma$ . We use  $f = \text{rec}(g_\varepsilon, \{g_a\}_{a \in \Sigma})$  as a shorthand for previous definition of recurrence. The following construction is redundant in presence of primitive recursion, but becomes essential when predicatively restricting it.

**Definition 24 (Case Distinction)** If  $g_\varepsilon : \mathbb{W}^k \rightarrow \mathbb{P}_{\mathbb{W}}$  and for every  $a \in \Sigma$ ,  $g_a : \mathbb{W}^{k+1} \rightarrow \mathbb{P}_{\mathbb{W}}$ , the function  $h : \mathbb{W}^{k+1} \rightarrow \mathbb{P}_{\mathbb{W}}$  such that  $h(\varepsilon, \mathbf{y}) = g_\varepsilon(\mathbf{y})$  and  $h(a \cdot w, \mathbf{y}) = g_a(w, \mathbf{y})$  is said to be defined by case distinction from  $g_\varepsilon$  and  $\{g_a\}_{a \in \Sigma}$  and is denoted as  $\text{case}(g_\varepsilon, \{g_a\}_{a \in \Sigma})$ .

In the following we will need also the following by definition of simultaneous recursion:

$\frac{}{\epsilon \triangleright \mathbb{W}_k \rightarrow \mathbb{W}_k}$	$\frac{}{c_a \triangleright \mathbb{W}_k \rightarrow \mathbb{W}_k}$	$\frac{}{r_a \triangleright \mathbb{W}_k \rightarrow \mathbb{W}_k}$	$\frac{}{\Pi_m^n \triangleright \mathbb{W}_{n_1} \times \dots \times \mathbb{W}_{n_n} \rightarrow \mathbb{W}_{n_m}}$
$\frac{\{g_i \triangleright \mathbb{W}_{s_1} \times \dots \times \mathbb{W}_{s_r} \rightarrow \mathbb{W}_{m_i}\}_{1 \leq i \leq l} \quad f \triangleright \mathbb{W}_{m_1} \times \dots \times \mathbb{W}_{m_p} \rightarrow \mathbb{W}_l}{f \odot (g_1, \dots, g_l) \triangleright \mathbb{W}_{s_1} \times \dots \times \mathbb{W}_{s_r} \rightarrow \mathbb{W}_l}$			
$\frac{g_\epsilon \triangleright \mathbf{W} \rightarrow \mathbb{W}_l \quad \{g_a \triangleright \mathbb{W}_k \times \mathbf{W} \rightarrow \mathbb{W}_l\}_{a \in \Sigma}}{\text{case}(g_\epsilon, \{g_a\}_{a \in \Sigma}) \triangleright \mathbb{W}_k \times \mathbf{W} \rightarrow \mathbb{W}_l}$		$\frac{g_\epsilon \triangleright \mathbf{W} \rightarrow \mathbb{W}_k \quad m > k \quad \{g_a \triangleright \mathbb{W}_k \times \mathbb{W}_m \times \mathbf{W} \rightarrow \mathbb{W}_k\}_{a \in \Sigma}}{\text{rec}(g_\epsilon, \{g_a\}_{a \in \Sigma}) \triangleright \mathbb{W}_m \times \mathbf{W} \rightarrow \mathbb{W}_k}$	

Figure 3: Tiering as a Typing System

**Definition 25** We say that the functions  $\mathbf{f} = (f^1, \dots, f^n)$  are defined by simultaneous primitive recursion over a word algebra  $\mathbb{W}$  from the function  $g_a^j$  (where  $j \in \{1, \dots, n\}$  and  $a \in \Sigma$ ) if the following holds for every  $j$  and for every  $a$ :

$$f^j(a \cdot v, \mathbf{w}) = g_a^j(f^1(v, \mathbf{w}), \dots, f^n(v, \mathbf{w}), v, \mathbf{w})$$

A function  $f^i$  as defined above will be indicated with  $\text{simrec}^i(\{g_\epsilon^j\}_j, \{g_a^j\}_{j,a})$ .

**Example 2** This definition allows to define, for instance, two functions  $f^0$  and  $f^1$  over a word algebra with  $\Sigma = \{a, b\}$ , as follows:

$$\begin{aligned} f^j(\epsilon, \mathbf{v}) &= g_\epsilon^j(\mathbf{v}) \quad \forall j \in \{0, 1\} \\ f^j(a \cdot w, \mathbf{v}) &= g_a^j(f^0(w, \mathbf{v}), f^1(w, \mathbf{v}), w, \mathbf{v}) \quad \forall j \in \{0, 1\} \\ f^j(b \cdot w, \mathbf{v}) &= g_b^j(f^0(w, \mathbf{v}), f^1(w, \mathbf{v}), w, \mathbf{v}) \quad \forall j \in \{0, 1\} \end{aligned}$$

### 3.1 Tiering as a Typing System

Now we define our type system which will then be used to introduce the definition of the class of *predicative probabilistic functions* and therefore to obtain our complexity result. The type system is inspired by the tiering approach of Leivant [12]. The idea behind tiering consists in working with denumerable many copies of the underlying algebra  $\mathbb{W}$ , each indexed by a natural number  $n \in \mathbb{N}$  and denoted  $\mathbb{W}_n$ . Type judgments take the form  $f \triangleright \mathbb{W}_{n_1} \times \dots \times \mathbb{W}_{n_k} \rightarrow \mathbb{W}_m$ , where  $f : \mathbb{W}^k \rightarrow \mathbb{W}$ . In the following, with slight abuse of notation,  $\mathbf{W}$  stands for any expression in the form  $\mathbb{W}_{i_1} \times \dots \times \mathbb{W}_{i_j}$ . Typing rules are given in Figure 3. The formulation of ramified recurrence over a probabilistic word algebra  $\mathbb{W}$  derives from the definition of recurrence, suitably restricted using types. The idea here is that, when generating functions by primitive recursion, one passes from a level (tier)  $m$  for the domain to a *strictly* lower level  $k$  for the result. This predicative constraint ensures that recursion does not causes complexity explosion.

Those probabilistic functions  $f : \mathbb{W}^k \rightarrow \mathbb{P}_{\mathbb{W}}$  such that  $f$  can be given a type through the rules in Figure 3 are said to be *predicatively recursive*. More precisely, the class  $\mathcal{PT}$  of all predicatively recursive functions is defined as follows.

**Definition 26** The class  $\mathcal{PT}$  of predicatively probabilistic recursive functions is the smallest class of functions that contains the basic functions and is closed under the operation of General Composition (Definition 4), Primitive Recursion (Definition 5), Case Distinction (Definition 24) and such that each function can be given a type through the rules in Figure 3.

Next we give the definition of the class of simultaneous recursive functions  $\mathcal{SR}$ .

**Definition 27** The class  $\mathcal{SR}$  of simultaneous recursive functions is the smallest class of functions that contains the basic functions and is closed under the operation of General Composition (Definition 4), Simultaneous Recursion (Definition 25), Case Distinction (Definition 24) and such

that each function can be given a type through the rules in Figure 3, plus the rule below:

$$\frac{\begin{array}{c} \{g_\epsilon^j \triangleright \mathbf{W} \rightarrow \mathbb{W}_k\}_j \quad m > k \\ \{g_a^j \triangleright \mathbb{W}_k^n \times \mathbb{W}_m \times \mathbf{W} \rightarrow \mathbb{W}_k\}_{j,a} \end{array}}{\text{simrec}^i(\{g_\epsilon^j\}_j, \{g_a^j\}_{j,a}) \triangleright \mathbb{W}_m \times \mathbf{W} \rightarrow \mathbb{W}_k}$$

### 3.2 Simultaneous Primitive Recursion and Predicative Recursion

We can encode Simultaneous Primitive Recursion in Predicative Recursion.

In fact, according to previous definition, if we have the two functions  $f^0, f^1$  over a word algebra with  $\Sigma = \{c_0, c_1\}$ , defined by simultaneous recursion as follows:

$$\begin{aligned} f^0(\epsilon, \mathbf{x}) &= g_\epsilon^0(\mathbf{x}) \\ f^0(c_0(w), \mathbf{x}) &= g_0^0(f(w, \mathbf{x})^0, f(w, \mathbf{x})^1, w, \mathbf{x}) \\ f^0(c_1(w), \mathbf{x}) &= g_1^0(f(w, \mathbf{x})^0, f(w, \mathbf{x})^1, w, \mathbf{x}) \\ f^1(\epsilon, \mathbf{x}) &= g_\epsilon^1(\mathbf{x}) \\ f^1(c_0(w), \mathbf{x}) &= g_0^1(f(w, \mathbf{x})^0, f(w, \mathbf{x})^1, w, \mathbf{x}) \\ f^1(c_1(w), \mathbf{x}) &= g_1^1(f(w, \mathbf{x})^0, f(w, \mathbf{x})^1, w, \mathbf{x}) \end{aligned}$$

we can see that we can define a function  $\tilde{f}: \mathbb{W}^{k+1} \rightarrow \mathbb{W}^2$  as follows:

$$\begin{aligned} \tilde{f}(\epsilon, \mathbf{x}) &= [g_\epsilon^1(\mathbf{x}), g_\epsilon^0(\mathbf{x})] \\ \tilde{f}(c_1(w), \mathbf{x}) &= [g_1^1(f(w, \mathbf{x})^0, f(w, \mathbf{x})^1, w, \mathbf{x}), g_1^0(f(w, \mathbf{x})^0, f(w, \mathbf{x})^1, w, \mathbf{x})] \\ \tilde{f}(c_0(w), \mathbf{x}) &= [g_0^1(f(w, \mathbf{x})^0, f(w, \mathbf{x})^1, w, \mathbf{x}), g_0^0(f(w, \mathbf{x})^0, f(w, \mathbf{x})^1, w, \mathbf{x})] \end{aligned}$$

Now the codomain of  $\tilde{f}$  can be coded in a single value of  $\mathbb{W}$ , this function is said *couple<sub>m</sub>*. Inversely we can define two functions, *first<sub>m</sub>* and *second<sub>m</sub>* that given a value in  $\mathbb{W}$  *first<sub>m</sub>* returns the first value of the couple, and the *second<sub>m</sub>* returns the second values.

Now we proof as this encoding uses at most  $m$  time the recurrence.

Firstly we define the length of a word as follows.

**Definition 28** We define  $|w|$  as the height of the parse-tree of the word  $w$ .

**Lemma 7** Let  $\mathbb{W}$  be word algebra,  $m > 0$ , and using at most 1 level of recurrence. There are functions (*couple<sub>m</sub>* :  $\mathbb{W} \times \mathbb{W} \rightarrow \mathbb{W}$ , *first<sub>m</sub>* :  $\mathbb{W} \times \mathbb{W} \rightarrow \mathbb{W}$ , *second<sub>m</sub>* :  $\mathbb{W} \times \mathbb{W} \rightarrow \mathbb{W}$ ) such that:

$$\begin{aligned} \text{couple}_m(u, v) &= t \\ \text{first}_m(t) &= u \\ \text{second}_m(t) &= v \end{aligned}$$

whenever  $2|u| + 2|v| + 2 \leq |t|^m$

### 3.3 Register Machines vs. Turing Machines

Register machines are a class of abstract computational model which, when properly defined, are Turing powerful. Here we extend the classical definition of Register Machines to the probabilistic case.

**Definition 29 (Probabilistic Register Machine)** *A Probabilistic Register Machine (PRM) consists of a finite set of registers  $\Pi = \{\pi_1, \dots, \pi_r\} = \pi$  which can store elements belonging to  $\mathbb{W}$  and of a finite sequence of indexed instructions (called program) which can have the following format:*

$\epsilon(\pi_s)(\pi_l)$

$c_i(\pi_s)(\pi_l)$

$p_i(\pi_s)(\pi_l)$

$jump(\pi_s)(\mathbf{m})$

$jump_{rand}(m)$

where  $\pi_s, \pi_l$  are registers,  $m$  is the number of the instruction and  $c_a$  is any constructor of the algebra  $\mathbb{W}$  and  $\mathbf{m}$  is a vector of  $|\Sigma|$  elements of natural number.

The semantic of previous instructions can be described as follows. We assume that the index of the current instruction is  $n$ .

$\epsilon(\pi_s)(\pi_l)$  is the  $\epsilon$  instruction, which stores in the register  $\pi_l$  the term resulting from the application of the constructor  $\epsilon$  to the register  $(\pi_s)$  and then transfer the control to the next instruction  $(n + 1)$ .

$c_a(\pi_s)(\pi_l)$  is the constructor instruction, which stores in the register  $\pi_l$  the term resulting from the application of the constructor  $c_a$  to the register  $(\pi_s)$  and then transfer the control to the next instruction  $(n + 1)$ .

$p_a(\pi_s)(\pi_l)$  is the predecessor instruction which stores in the register  $\pi_l$  the element resulting from the application of the predecessor  $p_a$  to the register  $\pi_s$ , and then transfer the control to the next instruction  $(n + 1)$ .

$jump(\pi_s)(\mathbf{m})$  is the jump instruction which: jumps to the instruction  $m_{\bar{a}}$  and stores the result of  $p_a(\pi_s)$  in  $\pi_s$  if apply  $c_a$  is the first constructor in the value contained in  $\pi_s$ ; transfer the control to the next instruction  $n + 1$  if  $\pi_s$  contains  $\epsilon$ .

$jump_{rand}(m)$  is the jump randomize instruction which jumps to the instruction  $m$  with probability  $1/2$ , or transfer the control to the next instruction  $n + 1$  with probability  $1/2$ .

Below we formalize more precisely the semantics of a PRM in terms of configurations which can be modified by the instructions. Hence we introduce the following,

**Definition 30 (Configuration of a PRM)** *Let  $R$  be a PRM be as in 29. We define a PRM configuration as a tuple  $\langle v_1, \dots, v_r, n \rangle$  where:*

- the  $v_i$ s are the values of the registers;
- $n$  is a natural number indicating the current instruction.

We define the set of all configurations with  $\mathcal{CR}_R$ . When  $n = 1$  we have an initial configuration for  $r$  strings  $\mathbf{s}$  is indicated with  $\mathcal{INR}_R^{\mathbf{s}}$ . When  $n = \max + 1$ , where  $\max$  is the number of instructions in the program, we have a final configuration said  $\mathcal{FCR}_R^{\mathbf{s}}$ .

Next we show how previous instructions allow to change a configuration.

$\epsilon(\pi_s)(\pi_l)$  If we apply the instruction  $\epsilon(\pi_s)(\pi_l)$  to the configuration  $\langle p_1, \dots, p_r, n \rangle$ , we obtain the configuration  $\langle p_1, \dots, p_{l-1}, p_s, p_{l+1}, \dots, p_r, n+1 \rangle$ .

$c_a(\pi_s)(\pi_l)$  If we apply the instruction  $c_a(\pi_s)(\pi_l)$  to the configuration  $\langle p_1, \dots, p_r, n \rangle$ , we obtain the configuration  $\langle p_1, \dots, p_{l-1}, a \cdot p_s, p_{l+1}, \dots, p_r, n+1 \rangle$ .

$p_a(\pi_s)(\pi_l)$  If we apply the instruction  $p_a(\pi_s)(\pi_l)$  to the configuration  $\langle p_1, \dots, a \cdot p_s, \dots, p_r, n \rangle$ , we obtain the configuration  $\langle p_1, \dots, a \cdot p_s, \dots, p_{l-1}, p_s, p_{l+1}, \dots, p_r, n+1 \rangle$ .

$jump(\pi_s)(\mathbf{m})$  If we apply  $jump(\pi_s), (\mathbf{m})$  to the configuration  $\langle p_1, \dots, a \cdot p_s, \dots, p_r, n \rangle$ , we obtain the configuration  $\langle p_1, \dots, p_s, \dots, p_r, m_{\pi} \rangle$ ; If we apply  $jump(\pi_s), (\mathbf{m})$  to the configuration  $\langle p_1, \dots, p_{s-1}, \epsilon, p_{s+1}, \dots, p_r, n \rangle$ , we obtain the configuration  $\langle p_1, \dots, p_{s-1}, \epsilon, p_{s+1}, \dots, p_r, n+1 \rangle$ .

$jump_{rand}(m)$  If we apply  $jump_{rand}, s$  to the configuration  $\langle p_1, \dots, p_r, n \rangle$ , we obtain the configuration  $\langle p_1, \dots, p_r, m \rangle$  with probability  $1/2$  and the configuration  $\langle p_1, \dots, p_r, n+1 \rangle$  with probability  $1/2$ .

Note that, according to the previous definition, one of the instructions of the machine is  $jump_{rand}$ , which gives to the machine the probabilistic behavior. So in order to simulate the behavior of a PTM we can assume that we apply  $jump_{rand}$  after any instruction of another type.

Intuitively, the function computed by a PRM  $R$  associates to each input  $s^r$  a (pseudo)-distribution which indicates the probability of reaching a configuration in  $\mathcal{FCR}_R^t$  from  $\mathcal{INR}_R^s$ . It is worth noticing that, differently from the deterministic case, since in a PRM the same final configuration can be obtained by different computations, the probability of reaching a given final configuration is the *sum* of the probabilities of reaching the configuration along all computation paths, of which there can be (even infinitely) many. So also in this case it is convenient to define the function computed by a PRM through a fixpoint construction, as follows. First we observe that the meaning of a PRM  $R$  program can be defined by using two functions  $\delta_0$  and  $\delta_1$ . In fact as previously mentioned we assume that in each PRM program we use  $jump_{rand}$  after any instruction of another type. Hence we can consider two functions  $\delta_0 : \mathcal{CR}_R \rightarrow \mathcal{CR}_R$  and  $\delta_1 : \mathcal{CR}_R \rightarrow \mathcal{CR}_R$  which, given a configuration in input, both produce in output the (unique) configuration resulting from the application of an instruction, when this is different from  $jump_{rand}$ . When, on the other hand,  $jump_{rand}$  is used,  $\delta_0$  and  $\delta_1$  produce respectively the two configurations (with probability  $1/2$ ) resulting from the two branches of the instruction, as previously defined.

We can define a (complete) partial order on the  $\mathcal{CEV}$  elements analogously to what we have done for the PTM. Hence we can now define a functional  $FR_R$  on  $\mathcal{CEV}$  which will be used to define the function computed by  $R$  via a fixpoint construction. Intuitively, the application of the functional  $FR_R$  describes *one* computation step. Formally we have the following:

**Definition 31** Given a PRM  $R$ , we define a functional  $FR_R : \mathcal{CEV} \rightarrow \mathcal{CEV}$  as:

$$FR_R(f)(C) = \begin{cases} \{s^1\} & \text{if } C \in \mathcal{FCR}_M^s; \\ \frac{1}{2}f(\delta_0(C)) + \frac{1}{2}f(\delta_1(C)) & \text{otherwise.} \end{cases}$$

Using similar arguments to those of Proposition 6 and Theorem 1 we can show that there exists the least fixpoint of the functional defined above. Such a least fixpoint, once composed with a function returning  $\mathcal{INR}_R^s$  from  $\mathbf{s}$ , is the *function computed by the register machine  $R$*  and it is denoted by  $\mathcal{IO}_R : \Sigma^* \rightarrow \mathbb{P}_{\Sigma^*}$ . The set of those functions which can be computed by any PRMs is denoted by  $\mathcal{PT}$ . Moreover, we denote the class of the function computed by a polynomial time register machine as  $\mathcal{PPR}$ .

Next Lemma shows the relations between PTMs and PRMs.

**Lemma 8** PTMs are linear time reducible to a PRMs, and PRMs over  $\mathbb{W}$  are poly-time reducible to PTMs.

**Proof.** A single tape PTM  $M$  can be simulated by a PRM  $R$  that has tree registers. A configuration  $\langle w, a, v, s \rangle$  of  $M$  can be coded by the configuration  $R \langle [w^r, a, v], s \rangle$  where  $s^r$  denotes the reverse of the string  $s$ . Each move of  $M$  is simulated by at most 2 moves of  $R$ . In order to simulate the probabilistic part given by the functions  $\delta_0$  and  $\delta_1$  we use an instruction  $jump_{rand}$  by assuming, for example, that if  $jump_{rand}$  allows to jump to  $m$  we simulate  $\delta_0$  (that is at the index  $m$  we have the PRM code simulating  $\delta_0$ ) otherwise we simulate  $\delta_1$ . Conversely, a PRM  $R$  over  $\mathbb{W}$  with  $m$  registers is simulated by an  $m$ -tapes PTM  $M$ . Some move of  $R$  may require copying the contents of one register to another for which  $M$  may need as many steps to complete as the maximum of the current lengths of the corresponding tapes. Thus  $R$  runs in time  $O(n^k)$ , then  $M$  runs in time  $O(n^{2k})$ .  $\square$

### 3.4 Poly-time Soundness

In this section we prove that any function definable by predicative recurrence is computable by a polynomial time probabilistic register machine said PPRM.

From the lemma 8 we can derive that  $\mathcal{PPR} = \mathcal{PPC}$ . Hence, in order to prove the equivalence result we first show that a predicative recurrence can be computed by a PPRM. This result is not difficult and is proved by exhibiting a PPRM which simulate the basic predicative recurrence functions and by showing that  $\mathcal{PPR}$  is closed by composition, primitive recursion, and case distinction, that is, we can construct a PPRM which simulates these operations (on the machine representing predicative recurrence functions).

We start with the following Lemmata.

**Lemma 9 (Basic Functions and PPRM Computability)** *All Basic Probabilistic Functions are computable by a PPRM.*

**Proof.** We need to show that for every basic function defined in Definition 2 we can construct a PPRM that computes such a function. The proof is immediate for functions,  $c_a$ , by observing that it is included in the set of PPRM operations. The function  $\epsilon$  is simulated by using the instruction  $\epsilon$  (on an empty register). The function  $\Pi$  is simulated by the instructions  $\epsilon(\pi_s)(\pi_t)$ . Finally the function  $rand$  can be simulated by the instructions  $jump_{rand}$  and  $c_a$ .  $\square$

**Lemma 10 (Generalized Composition and PPRM Computability)** *Given PPRM-computable  $f : \mathbb{N}^n \rightarrow \mathbb{P}_{\mathbb{N}}$ , and  $g_1 : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}, \dots, g_n : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$  the function  $f \odot (g_1, \dots, g_n) : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$  is itself PPRM computable*

**Proof.** We give an intuitive proof. We take a PPRM, said  $R_s$  with  $s$  registers. The first  $k$  registers have saved the input, the next  $k \cdot n$  registers  $R_s$  computes the  $g_1, \dots, g_n$  functions. For each  $g_i$  the machine saves the result on the registers  $(k \cdot n) + i$ , with  $1 \leq i \leq n$ . These registers became the input registers for computing  $f$ . Finally in the last register is saved the result.  $R_s$  operates as follows:

1.  $R_s$  copies all  $k$  registers on the  $k \cdot n$  registers. The computational cost of this operation is  $n \cdot k$ , because it is implemented by the instruction  $\epsilon(\pi_t)(\pi_s)$ ;
2.  $R_s$  computes the respective functions  $g_i$  with  $1 \leq i \leq n$  and saves the results in the registers  $(k \cdot n) + i$ . These functions are by hypothesis polynomial time computable;
3.  $R_s$  computes the function  $f$  that by hypothesis polynomial time computable.

Finally  $R_s$  computes the function  $f \odot (g_1, \dots, g_n)$  in time

$$\begin{aligned} z &= k \cdot n + \sum_{i=1}^n (\max(|pi_i|)^{s_i} + v_i) + \max(pi_{(k \cdot n) + i})^t + w \\ &\leq (k \cdot n) + (n + 1) \cdot (\max(|pi_i|, |pi_{(k \cdot n) + i}|)^{\max(s_i, t)} + \max(v_i, w)) \\ &< (n + 1) \cdot (y^m + q + k) \end{aligned}$$

where  $pi$  denotes the element saved on the register,  $\max(|pi_i|, |pi_{(k \cdot n) + i}|) = y$ ,  $\max(s_i, t) = m$  and  $\max(v_i, w) = q$ .  $\square$

**Lemma 11 (Case Distinction and PPRM-Computability)** *Given PPRM-Computable  $g_{\{a\}_{a \in \Sigma}} : \mathbb{N}^{k+2} \rightarrow \mathbb{P}_{\mathbb{N}}$ , and  $g_{\varepsilon} : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$ , the function  $\text{case}(g_{\varepsilon}, \{g_a\}_{a \in \Sigma})$  is itself PPRM-Computable.*

**Proof.** The function Case Distinction is implemented by a PPRM, said  $R_{\text{case}}$ , that computes it as follows. The first  $k+1$  registers contain the input and on the last register we have the result.  $R_{\text{case}}$  operates as follows:

1.  $R_{\text{case}}$  applies the operation  $\text{jump}(\pi_{k+1})(\mathbf{m})$ ; if  $pi_{k+1} = \epsilon$  the machine goes to the instruction  $i+1$  where there is saved the first instruction in order to compute  $g_{\varepsilon}$ , otherwise the machine jump at the instruction  $m_a$  corresponding at the first (in  $pi_{k+1}$ ) constructor function  $c_a$  and it saves the result on  $pi_{k+1}$  register. The instruction  $m_a$  is the first one which allows to compute  $g_a$ .
2.  $R_{\text{case}}$  computes the function  $g_{\varepsilon}$  or  $g_a$  and it saves the result on the last register.

Finally  $R_{\text{case}}$  computes the function Case Distinction in time  $z$ , and  $z$  is a polynomial time because:

$$z = c + \begin{cases} |s_{\varepsilon}|^{k_{\varepsilon}} + r_{\varepsilon} & \text{if } pi_{k+1} = \epsilon \\ |s_a|^{k_a} + r_a & \text{if } pi_{k+1} = p_a \end{cases}$$

where  $c$  is the time constant used from the machine for computing the function  $\text{jump}$ . □

**Lemma 12 (Primitive Recursion and PPRM-Computability)** *Given PPRM-Computable  $g : \mathbb{N}^{k+2} \rightarrow \mathbb{P}_{\mathbb{N}}$ , and  $f : \mathbb{N}^k \rightarrow \mathbb{P}_{\mathbb{N}}$ , the function  $\text{rec}(f, g) : \mathbb{N}^{k+1} \rightarrow \mathbb{P}_{\mathbb{N}}$  is itself PPRM-Computable.*

**Proof.** We give an intuitive proof. We take a PPRM, said  $R_{\text{rec}}$ . The first  $k+1$  registers contain the input, in the next  $k$  registers we save the input, in the  $2k+2^{\text{th}}$  register we have the result of intermediate computations, and in the last register we have the result. We assume that the  $k+1^{\text{th}}$  input is saved inverted on the  $k+1^{\text{th}}$  register.  $R_{\text{rec}}$  operates as follows:

1.  $R_{\text{rec}}$  computes  $g_{\varepsilon}$  and saves the results in the  $2k+2^{\text{th}}$ ;
2.  $R_{\text{rec}}$  applies an operation of  $\text{jump}(\pi_{k+1}), (\mathbf{m})$ ;
3. if  $pi_{k+1} = \epsilon$  the machine goes to the instruction  $i+1$  where it saves the result on the last registers and then stop, otherwise the machine jump to the instruction  $m_a$  corresponding at the function predecessor  $p_a$  and it saves the result on  $pi_{k+1}$  register.  $m_a$  is the first instruction needed in order to compute  $g_a$ .
4.  $R_{\text{rec}}$  jump at the instruction 2 if it is not stopped before.

Finally  $R_{\text{rec}}$  computes the function  $\text{rec}(f, g)$  in time  $z$ , and  $z$  is a polynomial time because:

$$\begin{aligned} z &= c \cdot |p_{k+1}| + |p_{k+1}| \cdot \begin{cases} |s_{\varepsilon}|^{k_{\varepsilon}} + r_{\varepsilon} & \text{if } pi_{k+1} = \epsilon \\ |s_a|^{k_a} + r_a & \text{if } pi_{k+1} = p_a \end{cases} \\ &\leq \max(|p_{k+1}|, |s_{\varepsilon}|, |s_a|) \cdot (c + \max(r_{\varepsilon}, r_a)) + \max(|p_{k+1}|, |s_{\varepsilon}|, |s_a|) \cdot \max(|p_{k+1}|, |s_{\varepsilon}|, |s_a|)^{\max(k_{\varepsilon}, k_a)} \\ &= x^t + x \cdot d \end{aligned}$$

where  $c$  is the time constant used from the machine for computes the function  $\text{jump}$ ,  $x = \max(|p_{k+1}|, |s_{\varepsilon}|, |s_a|)$ ,  $t = \max(k_{\varepsilon}, k_a) + 1$  and  $d = c + \max(r_{\varepsilon}, r_a)$ . □

**Lemma 13 (PPRM Computable Functions are Predicative Recurrence Functions)**  $\mathcal{PT} \subseteq \mathcal{PR}$

**Proof.** We have that is proved by Lemma 9, 10, 12 and 11. □

**Lemma 14 (PPTM Computable Functions are Predicative Recurrence Functions)**  $\mathcal{PT} \subseteq \mathcal{PC}$

**Proof.** We have that is proved by Lemma 13 and 8. □



### 3.5 Poly-time Completeness

In this section, one can give an embedding of any polynomial time probabilistic register machine into a predicatively recursive function, making use of simultaneous recurrence.

In order to do this firstly we give some lemma in order to construct the proof.

We start defining some predicatively recursive functions in order to simulate a single step of a PPRM.

**Lemma 15** *Let  $R$  be a PPRM with  $r$  registers over  $\mathbb{W}$ , there are functions  $\phi_0 : \mathbb{W}^k \rightarrow \mathbb{P}_{\mathbb{W}}, \dots, \phi_r : \mathbb{W}^k \rightarrow \mathbb{P}_{\mathbb{W}}$ , with tier at least one, such that if  $R$  has a transition rule from the configuration  $i$  then  $(\mathbf{pi}, n) \rightarrow_R (\mathbf{pi}, m)$  if and only if  $\phi_i(\mathbf{pi}, \bar{n}) = p'_i$  for all  $1 \leq i \leq r$  and  $\phi_0(\mathbf{u}, \bar{n}) = \bar{n}$*

**Proof.** We observe that at every step each register's value  $pi_i$  can assume only one of these values  $(c_a(pi_i), p_a(pi_i), \epsilon(pi_i))$ . We note that the natural number of the instruction can be creased or decreased of a finite number. Applying the instruction  $jump_{rand}$  we obtain a change in our configuration in the number of the next instruction and the relative probability. Now we show as the instructions of our program are mapped by one or more predicatively recursive functions.

- starting from the configuration  $\langle p_1, \dots, p_r, n \rangle$  and applying  $\epsilon(\pi_s)(\pi_l)$  we obtain  $\langle p_1, \dots, p_{l-1}, p_s, p_{l+1}, \dots, p_r, n \rangle$ . So it can be mapped by these functions  $\phi_s = \Pi_s^{r+1}$ ,  $\phi_l = \Pi_s^{r+1}$ , and  $\phi_0 = c_a \odot (\Pi_0^{r+1})$ ;
- starting from the configuration  $\langle p_1, \dots, p_r, n \rangle$  and applying  $c_a(\pi_s)(\pi_l)$ , we obtain the configuration  $\langle p_1, \dots, p_{l-1}, a \cdot p_s, p_{l+1}, \dots, p_r, n+1 \rangle$ , it can be mapped by these functions  $\phi_s = \Pi_s^{r+1}$ ,  $\phi_l = c_a \odot (\Pi_s^{r+1})$ , and  $\phi_0 = c_a \odot (\Pi_0^{r+1})$ ;
- starting from the configuration  $\langle p_1, \dots, a \cdot p_s, \dots, p_r, n \rangle$  and applying  $p_a(\pi_s)(\pi_l)$  we obtain  $\langle p_1, \dots, a \cdot p_s, \dots, p_{l-1}, p_s, p_{l+1}, \dots, p_r, n+1 \rangle$ , it can be mapped by these functions  $\phi_s = \Pi_s^{r+1}$ ,  $\phi_l = p_a \odot (\Pi_s^{r+1})$ , and  $\phi_0 = c_a \odot (\Pi_0^{r+1})$ ;
- starting from the configuration  $\langle p_1, \dots, a \cdot p_s, \dots, p_r, n \rangle$  and applying  $jump(\pi_s), (\mathbf{m})$  we obtain the configuration  $\langle p_1, \dots, p_s, \dots, p_r, m_{\bar{a}} \rangle$  we obtain the configuration  $\langle p_1, \dots, p_{s-1}, \epsilon, p_{s+1}, \dots, p_r, n+1 \rangle$  it can be mapped by these functions  $\phi_s = p_a \odot (\Pi_s^{r+1})$ , for each  $m_{\bar{a}}$  with a function  $\phi_0$  that if  $m_{\bar{a}} - n \geq 0$  has exactly  $m_{\bar{a}} - n$  concatenation of constructors otherwise it has exactly  $m_{\bar{a}} - n$  predecessor functions, finally each of these functions are composed with the function  $case(g_{\epsilon}, \{g_a\}_{a \in \Sigma})$ ;
- starting from the configuration  $\langle p_1, \dots, p_r, n \rangle$  and applying  $jump_{rand}$  we obtain  $\langle p_1, \dots, p_r, m \rangle$  with probability  $1/2$  and the configuration  $\langle p_1, \dots, p_r, n+1 \rangle$  with probability  $1/2$ , it can be mapped by the function  $rand$ , and  $case(g_{\epsilon}, \{g_a\}_{a \in \Sigma})$ . We note that in this case, the output of our register machine defines a distribution.

We note that all these functions are predicatively recursive functions, and every instructions are mapped with functions with at most flat recurrence, and so they can be mapped with tier 0.  $\square$

**Lemma 16** *If a function  $f$  over a word algebra  $\mathbb{W}$  is computable by a PPRM  $R$  in time  $t \leq Cn^k$ , then it is definable by  $k$  Simultaneous Recurrence Functions over  $\mathbb{P}_{\mathbb{W}}$  with tier at least one, applied to functions over  $\mathbb{P}_{\mathbb{W}}$*

**Proof.** We start our proof by considering the case  $C = 1$  and then we extend the proof to the case  $C > 1$ . We assume that  $R$  has  $r$  registers. Now we define the functions  $\sigma_{qj} : \mathbb{W}^{q+m+2} \rightarrow \mathbb{P}_{\mathbb{W}}$ , with  $0 \leq q \leq k$  and  $j = 0, \dots, r$ , with tier at most one. The functions  $(\sigma_{q1}, \dots, \sigma_{qr})$  represent the values of the registers after  $|y_1| \cdot \dots \cdot |y_q| + |x|$  execution steps of  $R$ , starting from the initial configuration. The function  $\sigma_{q0}$  represents the index of the instruction after  $|y_1| \cdot \dots \cdot |y_q| + |x|$  execution steps.

For each register and index instruction we define a function with at most  $k$  nested recursion as follows:

$$\begin{aligned}\sigma_{0j}(\epsilon, (\mathbf{u}, \overline{n})) &= (\mathbf{u}, \overline{n}) \\ \sigma_{0j}(a \cdot w, (\mathbf{u}, \overline{n})) &= \phi_j(\sigma_{0j}(\mathbf{u}, \overline{n})) \\ \sigma_{q+1,j}(\epsilon, \mathbf{y}, x, (\mathbf{u}, \overline{n})) &= \sigma_{0j}(w, x, (\mathbf{u}, \overline{n})) \\ \sigma_{q+1,j}(a \cdot w, \mathbf{y}, x, (\mathbf{u}, \overline{n})) &= \sigma_{q,j}(\sigma_{0j}(\mathbf{y}, x, (\mathbf{u}, \overline{n})), \sigma_{q+1}(w, \mathbf{y}, \epsilon, (\mathbf{u}, \overline{n})))\end{aligned}$$

Now it suffices to use simultaneous recurrence for composing the function define on every register and this prove the thesis.  $\square$

**Lemma 17 (Class of Simultaneous Recurrence Functions is the Class of Probabilistic Register Computable)**  
 $\mathcal{PPC} \subseteq \mathcal{SR}$

**Proof.** We have that is proved by Lemma 15, Lemma 16  $\square$

**Theorem 4 (Class of Predicative Probabilistic Functions is the Class of Probabilistic Register Computable)**  
 $\mathcal{PPC} = \mathcal{PT}$

**Proof.** We have that is proved by 13, Lemma 17 and Lemma 7.  $\square$

## 4 Conclusions

In this paper we make a first step in the direction of characterizing probabilistic computation in itself, from a recursion-theoretical perspective, without reducing it to deterministic computation. The significance of this study is genuinely foundational: working with probabilistic functions allows us to better understand the nature of probabilistic computation on the one hand, but also to study the implicit complexity of a generalization of Leivant's predicative recurrence, all in a unified framework.

More specifically, we give a characterization of computable probabilistic functions by a natural generalization of Kleene's partial recursive functions which includes, among initial functions, one that returns the uniform distribution on  $\{0, 1\}$ . We then prove the equi-expressivity of the obtained algebra and the class of functions computed by PTMs. In the the second part of the paper, we investigate the relations existing between our recursion-theoretical framework and sub-recursive classes, in the spirit of ICC. More precisely, endowing predicative recurrence with a random base function is proved to lead to a characterization of polynomial-time computable probabilistic functions.

An interesting direction for future work could be the extension of our recursion-theoretic framework to *quantum* computation. In this case one should consider transformations on Hilbert spaces as the basic elements of the computation domain. The main difficulty towards obtaining a completeness result for the resulting algebra and proving the equivalence with quantum Turing machines seems to be the definition of suitable recursion and minimization operators generalizing the ones described in this paper, given that qubits (the quantum analogues of classical bits) cannot be copied nor erased.

## References

- [1] S. Bellantoni and S. Cook. A new recursion-theoretic characterization of the polytime functions. *Computational complexity*, 2(2):97–110, 1992.
- [2] A. Bogdanov and L. Trevisan. Average-case complexity. *Foundations and Trends in Theoretical Computer Science*, 2(1), 2006.

- [3] U. Dal Lago and P. P. Toldin. A higher-order characterization of probabilistic polynomial time. In *Foundational and Practical Aspects of Resource Analysis*, pages 1–18. Springer, 2012.
- [4] U. Dal Lago and S. Zuppiroli. Probabilistic recursion theory and implicit computational complexity (long version). <http://eternal.cs.unibo.it/prtic.pdf>, 2014.
- [5] K. De Leeuw, E. F. Moore, C. E. Shannon, and N. Shapiro. Computability by probabilistic machines. *Automata studies*, 34:183–198, 1956.
- [6] J. Gill. Computational complexity of probabilistic Turing machines. *SIAM Journal on Computing*, 6(4):675–695, 1977.
- [7] J.-Y. Girard. Light linear logic. *Inf. Comput.*, 143(2):175–204, 1998.
- [8] O. Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, 2000.
- [9] S. Goldwasser and S. Micali. Probabilistic encryption. *Journal of computer and system sciences*, 28(2):270–299, 1984.
- [10] J. Katz and Y. Lindell. *Introduction to Modern Cryptography (Chapman & Hall/Crc Cryptography and Network Security Series)*. Chapman & Hall/CRC, 2007.
- [11] S. C. Kleene. General recursive functions of natural numbers. *Mathematische Annalen*, 112(1):727–742, 1936.
- [12] D. Leivant. Ramified recurrence and computational complexity i: Word recurrence and poly-time. In *Feasible Mathematics II*, pages 320–343. Springer, 1995.
- [13] D. Leivant and J.-Y. Marion. Lambda calculus characterizations of poly-time. *Fundam. Inform.*, 19(1/2):167–184, 1993.
- [14] M. O. Rabin. Probabilistic automata. *Information and control*, 6(3):230–245, 1963.
- [15] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM J. Res. Dev.*, 3(2):114–125, 1959.
- [16] E. S. Santos. Probabilistic Turing machines and computability. *Proceedings of the American Mathematical Society*, 22(3):704–710, 1969.
- [17] E. S. Santos. Computability by probabilistic turing machines. *Transactions of the American Mathematical Society*, 159:165–184, 1971.
- [18] R. I. Soare. *Recursively enumerable sets and degrees: a study of computable functions and computably generated sets*. Perspectives in mathematical logic. Springer-Verlag, 1987.